Intersection of Convex Objects in the Plane

Lukas Walther, 20107539

Master's Thesis, Computer Science
July 2015
Advisor: Gerth Stølting Brodal
Co-Advisor: Peyman Afshani

AARHUS UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

Many applications require intersection detection and several algorithms have been invented that determine if two convex polygons intersect. We discuss three such algorithms and present their methods in detail. The first algorithm has recently been presented by Barba and Langerman [1]. We expand upon their proofs and show that the algorithm works correctly with all possible inputs and provide many implementation details that are left out in the original paper. The second algorithm is that of Dobkin and Souvaine [7]. This algorithm requires the top most and bottom most vertices on polygons and we show how to find them in logarithmic time. Both algorithms make use of binary search and are $O(\log n)$ time algorithms, where n is the number vertices in the two polygons. The third algorithm is that of Chazelle, Liu and Magen [3], which uses $O(\sqrt{n})$ time and has relaxed requirements on its input. This allows polygons to be stored in linked lists, which are more convenient to use than sorted arrays, which are required by the first two algorithms. The algorithm makes use of randomization and linear programming. We provide the linear programming methods required by the algorithm to find separating lines between two sets of points. All algorithms are implemented and their efficiency is compared through experiments. We conclude that the algorithm of Barba et al. [1] is the most efficient algorithm, with the algorithm of Chazelle et al. [3] being a good alternative if more flexibility is required.

Acknowledgements

I would like to thank my advisors Peyman Afshani and Gerth Stølting Brodal for the inspiring and challenging discussions and their guidance throughout this project. I also would like to thank Helene Haagh, Nickals Pingel, and Martin Vang for reading through this thesis and providing valuable feedback. Thank you to Mathies Christensen, Yannik Faase, Rasmus Hallenberg-Larsen, Simon Nordved, Tom Quast, and Thomas Sandholt for several helpful discussions.

Lukas Walther,
Aarhus, July 9, 2015.

Contents

Abstract i

Acknowledgments iii

1 Introduction 3

1.1 Overview 4

2 Preliminaries 7

3 Algorithms 13

3.1 Barba and Langermans Algorithm 13

3.1.1 Constructions 14

3.1.2 Invariants 14

3.1.3 Overview of the Algorithm 15

3.1.4 Initializing the Algorithm 16

3.1.5 Determine the Invariant 16

3.1.6 Separation Invariant Subroutine 21

3.1.7 Intersection Invariant Subroutine 23

3.1.8 Constant Size 26

3.1.9 Running time and Correctness 27

3.1.10 Implementation Details 27

3.2 Dobkin and Souvaine's Algorithm 29

3.2.1 Polygonal Chains 29

3.2.2 Binary Search for Extremum Points 30

3.2.3 Binary Search on Polygonal Chains 34

3.2.4 Pruning Two Chains 35

3.2.5 Pruning the Remaining Chain 37

3.2.6 Complete Algorithm 38

3.2.7 Implementation Details 39

3.3 Sublinear Algorithm 42

3.3.1 Storage 42

3.3.2 Algorithm 43

3.3.3 Implementation Details 44

3.3.4 Linear Programming 45

3.3.5 Implementation Details for Linear Programming 49

4 Experiments **51**
4.1 Setup 51
4.1.1 Starting Point for Generating Inputs 52
4.2 Comparing the algorithms of Dobkin and Barba 52
4.2.1 First Experiment 53
4.2.2 Second Experiment: Iterations 55
4.2.3 Third Experiment 56
4.2.4 Fourth Experiment: Non-uniformly Distributed Vertices . 56
4.3 Running time for the Sublinear Algorithm 58
4.3.1 Fifth Experiment: Verify the Asymptotic Running Time . 58
4.4 Comparing the Algorithms. 58
4.5 Concluding the Experiments 59

5 Conclusion **61**
5.1 Future Work 62

Bibliography **62**

List of Figures

2.1 Sidedness 8

3.1 The shapes $\mathcal{T}_P$ and $\mathcal{T}_Q$ 14
3.2 Intersection point between $\mathcal{T}_P$ and $\mathcal{T}_Q$ 17
3.3 Detecting intersections between $\mathcal{T}_P$ and $\textsc{ch}(e_0, e_1, e_2)$ 18
3.4 The two shapes of $\mathcal{T}_Q$ 20
3.5 Illustrating the separating invariant subroutine 22
3.6 Illustrating the intersection invariant subroutine 24
3.7 Detecting intersection when a vertex of $\mathcal{T}_P$ is inside a disjoint region. 24
3.8 $\mathcal{T}_P$ inside $\mathcal{T}_Q$ 27
3.9 Polygonal chains of P 30
3.10 Upward direction 31
3.11 All upward directions are equal 32
3.12 Illustrating the LR-region 34
3.13 Position of the LR-region 35
3.14 Using a separating line to construct parts of P's boundary 44
3.15 Linear programming 47

4.1 Generating random polygons on an ellipse 53
4.2 Running times of Barba's and Dobkin's algorithms 54
4.3 Number of iterations for Barba's and Dobkin's algorithms 55
4.4 Piecewise linear probability functions 57
4.5 Randomly generated polygons 57
4.6 Running time of the sublinear algorithm 59
4.7 Comparing the running times of the three algorithms 60
4.8 Comparing the sublinear algorithm with the naive solution . . . 60

Chapter 1

Introduction

Detecting the intersection between geometric objects is an important subject of study in the field of computational geometry and several algorithms have been proposed to solve these kind of problems. The scope of this thesis is to detect intersection between two convex objects in the plane. This kind of intersection detection has various application in for example computer graphics, motion planning and hit detection. A solution to this problem that uses linear time has been publicised by Shamos [11], who also introduced the well-known plane-sweep technique. Another linear time algorithm was later presented by O'Rourke et al. [9], which Saab improved upon [10]. Algorithms that provide logarithmic running time were studied by Chazelle and Dobkin [2] and alternative solutions with the same running time were presented by Dobkin and Kirkpatrick [6]. Later Dobkin and Souvaine [7] revisit those results and generalize the problem to curved polygons. Barba and Langerman [1] have recently presented a conceptually more simple algorithm as an introduction to concepts that they apply to convex shapes in three dimensions. Using a less restrictive set of assumptions about how the input to such algorithms is represented, Chazelle, Liu and Magen [3] present a randomized algorithm that uses sublinear time.

Published articles always provide an overview of the algorithms they present, but often leave many details out. This leaves a lot of work for the reader, if he wishes to implement those algorithms, which can be a difficult task. It is rare that published articles provide source code that implements the presented algorithms. The paper of Dobkin et al.[7] is an exception to this. They provide portions of the source code, but the code is rather concise and therefore difficult to understand. Thus implementing any algorithm that deals with the intersection between two convex polygons is not trivial and it would be valuable to have reference implementation as a starting point for future modification and optimization.

Goals The aim of this master thesis is to provide all details that are necessary to derive practical implementations of algorithms that determine if two convex polygons intersect. The first algorithm we present was presented by Barba et al. [1]. The algorithm is presented as an alternative to the algorithm of Dobkin

et al. [7], which is the second algorithm that we present. We have chosen to look at these algorithms, because the paper of Dobkin et al.[7] is a refinement of the methods first provided by Chazelle and Dobkin[2] and inherently different from the methods provided by Barba et al. [1]. The asymptotic running time of both algorithms is $O(\log n)$ without the need for preprocessing, which makes them inherently comparable and it would be valuable to know which algorithm to use in practice. Furthermore, we have chosen to look at the algorithm presented by Chazelle et al. [3] because of the less restrictive requirements about the input to the algorithm, making it useful in situations where more flexibility is required. We have chosen not to include any of the early algorithm by Shamos [11] or O'Rourke [9], because their linear time requirements makes it very unlikely that they can compete with their successors.

The accompanying implementation is done in `C++11` and can be found at `dl.dropboxusercontent.com/u/11699219/MasterThesis/PolygonIntersection.zip`. We invite the reader to follow along in the source code by using the footnotes provided throughout the thesis.

1.1 Overview

In this section, we provide a short overview for all chapters.

Preliminaries. We present concepts and methods that are used throughout the thesis, some of which are commonly found in the literature of Computational Geometry. Here we explain how we traverse and store polygons and how to find the median vertex between two other vertices on the boundary of a polygon. We also explain our conventions about line segments, intersection between them and how to determine where objects are in relation to them.

Algorithms. This chapter contains the detailed descriptions of the three algorithms that determine if two polygons P and Q intersect. We also provide lemmas and proof for details that are not covered in the articles that originally presented those algorithms.

The first part of the chapter covers an algorithm presented by Barba et al [1]. This algorithm requires the polygons to be represented with arrays, in which elements are stored in order. The algorithm can be labelled as a binary search algorithm, because it prunes a fraction of the input in each iteration, such that it has an asymptotic running time of $O(\log n)$. This is achieved by representing polygon P as a triangle that is contained in P, and representing Q as an edge hull, a shape that is bound by the lines extending three of Q's edges and that contains Q. If the triangle and the edge hull are separate, the algorithm will prune a part of P and choose a new triangle to represent the remaining parts of P. If the triangle and the edge hull are found to intersect, the algorithm either detects that P and Q intersect or prunes a part of Q and chooses a new edge hull to represent the remainder of Q. This is done until the

problem is reduced to constant size, at which point a constant time algorithm solves the problem.

The second part of the chapter covers an algorithm presented by Dobkin et al [7]. This algorithm also requires P and Q to be represented as an array, in which elements are stored in order. The algorithm performs binary search twice, resulting in an asymptotic running time of $O(\log n)$. This is achieved by splitting P and Q at their top most and bottom most vertices, resulting in four polygonal chains. Two pairs of these chains are then searched and pruned until an intersection between them is detected or they are reduced to constant size. If both pairs of chains are found to intersect, then P and Q intersect. Otherwise P and Q are separate.

The third and last part of the chapter covers an algorithm presented by Cazelle et al [3]. This algorithm is much different that the previous two, in that it only requires the input to be stored in linked lists, that are organized in successive memory. This less restrictive requirement means that it is not possible to do binary search on the boundary of the polygons, but it is possible to take a random sample from P and Q. The algorithm has an expected asymptotic running time of $O(\sqrt{n})$. The algorithm first takes a sample of P and Q. It then determines if the samples are separate, by using linear programming to find a separating line that is tangent to the samples. If the samples are not separate, then P and Q intersect. If the samples are separate, the algorithms constructs part of P's and Q's boundary, that are on the opposing sides of the separating line as the samples. Using these parts on the boundary, the algorithm uses linear programming several times to determine if there are any parts of P and Q that intersect.

Experiments. In this chapter, we compare the performance of the three algorithms, by measuring the time and number of iteration it takes to determine if to convex polygons intersect. We explain how we generate polygons that are used as input for the experiments. Furthermore, we explain how and why we change the way input is generated, and discuss the results.

Conclusion. In this chapter we will conclude the thesis by summing up the major points of the algorithms and summing up the results of the experiments. The chapter ends with a discussion of possible future work.

Chapter 2

Preliminaries

In this section, we present concepts that are used throughout this thesis. We introduce concepts from computation geometry such as shapes and convexity, but also details that are relevant to our implementation, specifically how polygons are stored and treated and how line segments and edges are stored and their intersection is computed.

Coordinate system. The coordinate system we use throughout this thesis is a 2 dimensional left handed coordinate system, such that the positive x-axis points to the right and the positive y-axis points upward.

Shape. A *shape* is a closed continuous set of points in the plane.

Intersection. Two shapes A and B *intersect* if $A \cap B \neq \emptyset$, where $\cap$ is the intersection defined by set theory, i.e. if a point is an element in both shapes, then it is an element in the intersection of those shapes.

Line Segment. A line segments is a shape that consists of the points on the shortest path from one point to another. We use two points to represent line segments. The line segment L can be parametrized as $L = \{a + s(b - a) \mid s \in [0, 1]\}$, where a and b are points in the plane and s is a scalar. To give a line segments a notion of direction, we say that a is the origin, or starting point, of the line segment, and the line segment goes from a to b. In this thesis, we write $l = (a, b)$ to denote that line segment l starts at a and ends at b. We also use $l.start$ and $l.end$ to denote the start and end point of l.

Convex Shape. A shape is convex, if for each pair of points in the shape, the line segment between the two points is also contained in the shape.

Intersection between line segments. We find the intersection between two line segments using the following method. Let $l = (a, b)$ and $m = (c, d)$ be directed line segments. We write l and m as

$$l : \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a.x \\ a.y \end{pmatrix} + s \begin{pmatrix} b.x - a.x \\ b.y - a.y \end{pmatrix} \tag{2.1}$$

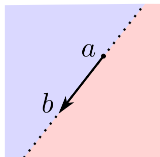


Figure 2.1: Sidedness. The directed line segment (a, b) in the plane. The red region contains all points that are to the left of (a, b), while the blue regions contains all points that are to the right of (a, b). The dotted line extends the directed line segment and all points on it are also considered to be on the infinite line that extends (a, b).

$$m : \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} c.x \\ c.y \end{pmatrix} + t \begin{pmatrix} d.x - c.x \\ d.y - c.y \end{pmatrix}. \tag{2.2}$$

Now s and t can be found using linear algebra, because we have four equations and two unknown variables. The following can be derived:

$$D = (d.x - c.x)(b.y - a.y) - (b.x - a.x)(d.y - c.y) \tag{2.3}$$

$$s = \frac{(d.x - c.x)(c.y - a.y) - (c.x - a.x)(d.y - c.y)}{D} \tag{2.4}$$

$$t = \frac{(b.x - a.x)(c.y - a.y) - (c.x - a.x)(b.y - a.y)}{D}, \tag{2.5}$$

as long as *denom* $\neq 0$, which is the case if the lines are not parallel. Now l and m intersect if and only if $s, t \in [0, 1]$. If $s < 0$ (resp. $t < 0$), then we say that the intersection would occur *before* the line segment l (resp. m). If $s > 1$ (resp. $t > 1$), then we say that the intersection would occur *after* the line segment l (resp. m). [1]

Sidedness A point v in the plane can either lie to the left of, to the right of, or on a directed line segment (a, b). To determine on which side of the line segment v is, we use the following procedure. Let $d = b - a$ and $e = v - a$. Let $\hat{d}$ be the orthogonal vector of d. If $\hat{d} \cdot e$ is positive, the point lies to the left of the line segment. If the dot product is negative, the point lies to the right of the line. Otherwise, the point is on the infinite line that contains the line segment. The sidedness of a line segment is illustrated in Figure 2.1.

Right turn. Three points a, b, and c form a right turn, if c is to the right of the directed line segment (a, b). Likewise, a left would be that c is the the left of the directed line segment (a, b).

Vertex. A *vertex* is a point in 2 dimensional space, that is the corner of a shape, like a triangle or polygon.

[1] Line 50-75 in `Line2f.cpp`

Edge. An *edge* is a line segment that connects two vertices. The notation that is used to denote line segments is also used for edges.

Convex Hull. The *convex hull* of a set of points is the the smallest convex shape that contains those points. Likewise, the convex hull of any of number of objects in the plane is the smallest convex shape that contains those objects. We denote the convex hull of any points or shapes with $\text{CH}(a, ..., b)$. For example, the convex hull of points p_0, p_1, and p_2 is $\text{CH}(p_0, p_1, p_2)$.

Polygons. A *polygon* is a shape that is bounded by a finite number of edges and has no holes in its boundary. The polygons we work with in this thesis are exclusively convex, which means that each point in the polygon is to the right of the polygons edges or contained in an edge. They are also simple polygons, which means that their boundaries do not intersect themselves. We define polygons to contain at least three such edges, such that they can not be single points or line segments.

Degenerate polygons. To reduce some of the complexity inherent in implementing algorithms that deal with geometric shapes, we disallow certain polygons, which we will call degenerate polygons. Degenerate polygons are polygons that contain three or more consecutive vertices that form a single line. This case could easily be avoided by removing the vertices in the middle, until only the outermost two vertices are left. Polygons are also degenerate if they contain two or more equal vertices, which would make it impossible to properly define line segments between them.

Representing and storing polygons. Polygons are usually represented by vertices that are connected through edges along the boundary of the polygon, such that we can easily identify neighbouring vertices. These vertices can be stored in arrays where neighbouring vertices are neighbours in the arrays, or in doubly linked lists, where each vertex also has a pointer to its neighbouring vertices. We let $size(P)$ denote the number of vertices in the some polygon P.

If polygons are stored in an array, we store them such that the vertices are in clockwise order, where three consecutive vertices a,b and c must form a right turn and consecutive entries in the array are neighbours on the boundary of the polygon. The first and last vertices in the array are considered neighbours, such that the element clockwise to the last element in the array is the first element. When using arrays, we use $i.x$ and $i.y$ to denote the x and y coordinates of the vertex at index i.

If polygons are stored in doubly linked lists, then each vertex is stored together with a pointer to its clockwise neighbour and a pointer to its counter clockwise neighbour.

Traversing arrays. A convex polygon that is stored in an array can be traversed in clockwise and counter clockwise direction. Let the array have n entries, where $n = size(P)$. Assume we have an index $0 \leq i < n$ and want the indices

that are neighbour to i. The clockwise index is $i+1$, if $i < n-1$, and 0 otherwise. The counter clockwise index to i is $i-1$ if $i > 0$ and $n-1$ otherwise.

Let $clock(i, P)$ denote the index clockwise to i, in the array that stores shape P and let $counter(i, P)$ denote the index counter clockwise to i, in the array that stores polygon P. It always holds, that $clock(counter(i, P), P) = i$.

Polygonal Chains. A polygonal chain is a sequence of vertices that are connected to each other by edges. The indices of the vertices on a closed polygonal chain of a polygon P are defined to be

$$chain(i, j, P) = \begin{cases} \{i, \dots, j\} & \text{if } i \leq j \\ \{0, \dots, j, i, \dots, size(P) - 1\} & \text{else} \end{cases},$$

where i and j are indices in the array that stores P. These are the indices that are encountered, when traversing an array in clockwise direction starting at i, until index j is encountered. We account for the fact that P is a closed polygon, by considering index 0 to be clockwise to index $size(P) - 1$, leaving no holes in the boundary of P.

Choosing median elements on a polygonal chain. We need to be able to choose median elements on polygonal chains, in order to do binary search. The middle index of $chain(a, b, P)$ can be found in the following way. Let $n = size(P)$. Let the distance between between a and b be $|chain(a, b, P)| - 1$, which is the number of clockwise steps we have to take to reach b from a. There are two possibilities for how a and b can relate to each other. If $a < b$, the distance between the two indices is $b - a$ and the middle index is $a + \frac{b-a}{2}$, i.e. we walk $\frac{b-a}{2}$ clockwise steps from a. If $b < a$, the distance between the indices is $b + (n - a)$, i.e. the we walk clockwise from a to $n - 1$ and from 0 to b. If $a + \frac{b+(n-a)}{2} < n$, the middle index is $a + \frac{b+(n-a)}{2}$, i.e. we walk half the distance from a. Otherwise, the middle index is $b - \frac{b+(n-a)}{2}$, i.e. we walk half the distance counter clockwise from b. This will be a valid index, because

$$\begin{aligned} a + \frac{b + (n - a)}{2} &\geq n \Leftrightarrow \\ a + b + n &\geq 2n \Leftrightarrow \\ b &\geq n - a \Leftrightarrow \\ 2b &\geq b + n - a \Leftrightarrow \\ b &\geq \frac{b + n - a}{2} \Leftrightarrow \\ b - \frac{b + n - a}{2} &\geq 0. \end{aligned}$$

Supporting Halfplanes. Let an edge e be part of the boundary of some convex polygon P. Unless otherwise noted, an edge is directed, such that its endpoint is the clockwise neighbour to its starting point on the boundary of P.

Consider the line ℓ to be the infinite line that extends and contains e. The line ℓ splits the plane in two halfplanes, that are complement to each other. We use ℓ^+ to denote the halfplane that contains all points that are to the right of e, and use ℓ^- to denote the halfplane that contains all points to the left of e. The halfplane ℓ^+ is said to support P, because ℓ^+ contains P. We also say that e supports ℓ and that ℓ is the line that extends e.

Separation. A line *separates* two objects in the plane, if the two object lie in the opposite halfplanes that the lines splits the plane into. A point separates two objects in the plane, if the horizontal line containing the point separates the objects.

Chapter 3

Algorithms

We present the details necessary to implement three algorithms that determine if two polygons P and Q intersect. The first algorithm is presented by Barba and Langerman [1]. The second algorithm is presented by Dobkin and Souvaine [7]. Both algorithms work on polygons that are stored in sorted clockwise order in an array and run in $O(\log n)$ time. The third algorithm is presented by Chazelle, Liu and Magen [3] and works on polygons that are stored in doubly linked lists and runs in $O(\sqrt{n})$.

3.1 Barba and Langermans Algorithm

Given two convex polygons P and Q, we want to determine whether they intersect. The polygons are given as two arrays, with their vertices stored in clockwise order.

The algorithm chooses three vertices from each polygon. These vertices are then used to represent the polygons in simplified form. The algorithm uses two different subroutines that each prune a fraction of one of the inputs. The process is repeated, until the polygons are found to intersect, are found to have a separating line, or are pruned to constant size. This process prunes a constant fraction of the polygons in each iterations and runs in $O(\log n)$ time. If both polygons are pruned to constant size, a constant time algorithm is used to determine intersection.

The description of this algorithm given by Barba et al. [1] explains the outlines of the algorithm, but does not provide all the details that are necessary to actually implement it. They also show that their algorithm is correct by using an invariant, but this invariant does not cover all cases. We expand upon the invariant and the proofs presented by Barba et al. and present the necessary details to implement the algorithm. Our implementation of the algorithm can be found in `intersectionBarba.cpp`.

We first present the constructions and invariants that Barba et al. use and introduce our own correctness invariant to show that their algorithm works. We then present an overview over the algorithm, show how to initialise it, and how to determine which subroutine to use. The two subroutines are the separation invariant subroutine and the intersection invariant subroutine, which are

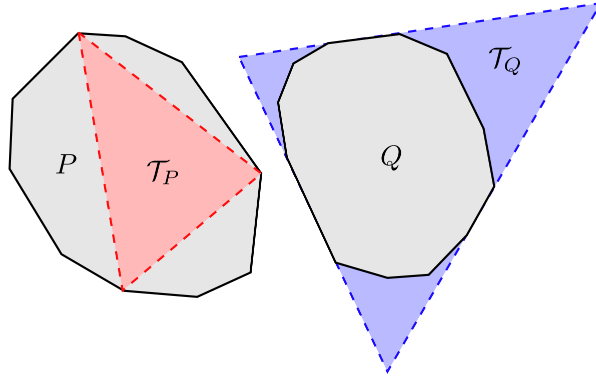


Figure 3.1: The convex polygons P and Q. The triangle $\mathcal{T}_P$ is defined by three vertices of P and is completely contained in P. The edge hull $\mathcal{T}_Q$ is defined as the intersection of the supporting half planes of three edges of Q. Polygon Q is completely contained in $\mathcal{T}_Q$.

explained in detail afterwards. We then show the constant time algorithm that is used at the very end of the algorithm and discuss additional implementation details at the end of Section 3.1.

3.1.1 Constructions

Vertices and edges. Let $V(P)$ and $E(P)$ (respectively $V(Q)$ and $E(Q)$) be the set of vertices and edges of P (resp. Q). Let $V^*(P)$ and $E^*(Q)$ be the sets of vertices and edges that remain after pruning steps are performed by the algorithm. Initially, $V(P) = V^*(P)$ and $E(Q) = E^*(Q)$.

Edge hull and triangle. Given some subset of edges $F \subseteq E(Q)$, we define the *edge hull* of F to be the intersection of the supporting halfplanes of each edge in F. Let $\mathcal{T}_Q$ be the edge hull of three edges of Q. This edge hull may be a triangle or a semi infinite region, see Figure 3.4. We denote the three edges that define $\mathcal{T}_Q$ with e_0, e_1, and e_2. Let $\mathcal{T}_P$ be the convex hull of three vertices on the boundary of P. This means that $\mathcal{T}_P$ will always be a triangle. The constructions $\mathcal{T}_P$ and $\mathcal{T}_Q$ are illustrated in Figure 3.1. We denote the three vertices that define the triangle as v_0, v_1, and v_2. Note that $\mathcal{T}_P \subseteq P$ and $Q \subseteq \mathcal{T}_Q$.

Disjoint regions. The edge hull $\mathcal{T}_Q$ can be split in up to three disjoint regions, by subtracting $\textsc{ch}(e_0, e_1, e_2)$ from $\mathcal{T}_Q$. These are the shapes r_0, r_1, and r_2 shown in Figure 3.3(b). We will refer to these as the disjoint regions of $\mathcal{T}_Q$.

3.1.2 Invariants

Barba et al. state two invariants. At least one of the invariants has to hold at the end of each iteration of the algorithm. They are only considered invariants in their respective subroutine, not invariant throughout the complete algorithm.

Invariant 1 (Separation invariant)**.** *"The separation invariant states that we have a line ℓ that separates $\mathcal{T}_P$ from $\mathcal{T}_Q$ such that ℓ is tangent to $\mathcal{T}_P$ at vertex v". [1, Section 2, The 2D Algorithm].*

Invariant 2 (Intersection invariant)**.** *"The intersection invariant states that we have a point in the intersection between $\mathcal{T}_P$ and $\mathcal{T}_Q$." [1, Section 2, The 2D Algorithm].*

Each of these two invariants is associated with a subroutine. The subroutine for Invariant 1 is presented in Section 3.1.6 and the subroutine for Invariant 2 is presented in Section 3.1.7. Barba et al. state that it is possible for both invariants to hold at the same time if $\mathcal{T}_P$ is tangent to $\mathcal{T}_Q$. They state that if both invariants hold, the subroutine can be chosen arbitrarily [1]. To prove that the their method works correctly, they state the following invariant:

> "[T]he *correctness invariant* [...] states that an intersection between P and Q can be computed with the remaining vertices and edges after the pruning. That is, P and Q intersect if and only if $\text{CH}(V^*(P))$ intersects an edge of $E^*(Q)$, where $\text{CH}(V^*(P))$ denotes the convex hull of $V^*(P)$." [1, Section 2, Algorithm in the plane].

We have defined intersection as $P \cap Q \neq \emptyset$, which occurs in three cases. Either the boundaries of P and Q intersect, P is completely inside Q or Q is completely inside P. According to their correctness invariant, intersection does only occur when the boundaries of P and Q intersect or Q is inside P, but if P is inside Q, no edge of Q would intersect P. Thus, our notion of intersection fundamentally disagrees with how the correctness invariant is stated. To resolve this issue, we extends the proofs provided by Barba et al. to work with the following correctness invariant:

Invariant 3 (Correctness Invariant)**.** *P and Q intersect if and only if $\text{CH}(V^*(P)) \cap \text{CH}(E^*(Q)) \neq \emptyset$.*

At the end of the Section 3.1.7 and Section 3.1.6, we argue correctness by showing that Invariant 3 holds after each iteration of the algorithm.

3.1.3 Overview of the Algorithm

The algorithm can be split in to five parts, which we will present in the following sections. We start by initializing $\mathcal{T}_P$ and $\mathcal{T}_Q$. Then, as long as both $V^*(P)$ and $E^*(Q)$ have more than a constant number of elements left, we determine which of the two invariants hold an use the associated subroutine, which reduces either $V^*(P)$ or $E^*(Q)$ in size. If there are only a constant number of elements left, we can determine if $\text{CH}(V^*(P))$ and $\text{CH}(E^*(Q))$ intersect in constant time. The following sums up the different parts of the algorithm.

- Initializing $\mathcal{T}_Q$ and $\mathcal{T}_P$.
- Determining which of Invariant 1 or Invariant 2 holds.
- The subroutine for Invariant 1.

- The subroutine for Invariant 2.
- The subroutine for constant size input.

Intuitively, when the intersection invariant holds, we choose an edge of $E^*(Q)$ that is closer to $\mathcal{T}_P$, such that $\mathcal{T}_Q$ would move away from $\mathcal{T}_P$ and remove elements from $E^*(Q)$ that are not necessary to detect intersection between P and Q. When the separation invariant holds, we choose a vertex of $V^*(P)$, such that $\mathcal{T}_P$ moves towards $\mathcal{T}_Q$ and remove elements from $V^*(P)$ than can not possibly intersect with Q.

3.1.4 Initializing the Algorithm

Initializing $\mathcal{T}_P$. Let $n = size(P)$. Set $v_0 = 0$, $v_1 = \lfloor n/3 \rfloor$, and $v_2 = \lfloor 2n/3 \rfloor$ to be indices in the array of P. Let $\mathcal{T}_P$ be the triangle constructed from the three directed line segments $t_0 = (v_0, v_1)$, $t_1 = (v_1, v_2)$ and $t_2 = (v_2, v_0)$. For ease of notation, we define $clock(v_i, \mathcal{T}_P) = v_{(i+1 \mod 3)}$ and $counter(v_i, \mathcal{T}_P) = v_{(i-1 \mod 3)}$ i.e. the $clock(v_i, \mathcal{T}_P)$ and $counter(v_i, \mathcal{T}_P)$ functions always return one of v_0, v_1, and v_2.

Initializing $\mathcal{T}_Q$. Let $m = size(Q)$. Set $w_0 = 0$, $w_1 = \lfloor m/3 \rfloor$, and $w_2 = \lfloor 2m/3 \rfloor$ to be indices in the array of Q. Then construct the three directed line segments

- $e_0 = (w_0, clock(w_0, Q))$
- $e_1 = (w_1, clock(w_1, Q))$
- $e_2 = (w_2, clock(w_2, Q))$.

Let $\mathcal{T}_Q$ be the edge hull defined by e_0, e_1, and e_2. For ease of notation, we define $clock(e_i, \mathcal{T}_Q) = e_{(i+1 \mod 3)}$ and $counter(e_i, \mathcal{T}_Q) = e_{(i-1 \mod 3)}$ [1].

Indices and pruning. Every time a subroutine is used, some vertices of P or Q are pruned. The remaining vertices and edges are denoted $V^*(P)$ and $E^*(Q)$. To clarify what elements of P are pruned and what elements are still present, we maintain v_0 and v_2 in the following way at the end of the separation invariant subroutine: If elements of P have been pruned, then the remaining indices are $V^*(P) = chain(v_0, v_2, P)$. Note that v_0 and v_2 are considered neighbours on the boundary $V^*(P)$ in this case.

To clarify what elements of Q are left after pruning, we maintain e_0 and e_2 at the end of the intersection invariant subroutine in the following way: If elements of Q have been pruned, then the remaining indices are $V^*(Q) = chain(e_0.start, e_2.end, Q)$ and $E^*(Q)$ is the set of edges that connects the vertices $V^*(Q)$ in clockwise order along the boundary.

3.1.5 Determine the Invariant

Given the edges that represent $\mathcal{T}_Q$ and the vertices that represent $\mathcal{T}_P$, we have to determine which of the separation and intersection invariant holds. To do

[1] Line 58 - 95 in `intersectBarba.cpp`

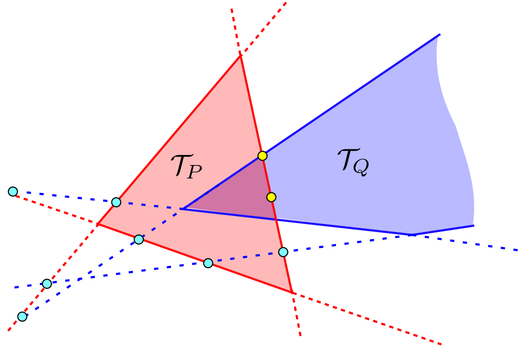


Figure 3.2: When searching for an intersection point between the boundaries between $\mathcal{T}_P$ and $\mathcal{T}_Q$, we have to consider all intersection points between all edges of $\mathcal{T}_P$ and $\mathcal{T}_Q$. The yellow dots are intersections that occur on the boundaries of $\mathcal{T}_P$ and $\mathcal{T}_Q$, while cyan dots occur outside of at least one of the shapes.

this, we search for an intersection point between an edge of $\mathcal{T}_P$ and an edge of $\mathcal{T}_Q$. This means that we do nine intersection computations in total, determining intersection between every e_i and every t_j, where $i, j \in \{0, 1, 2\}$, that we have constructed previously [2]. For every resulting intersection point, we test if the intersection occurs on t_j and if so, if the point is to the right of the two edges of $\mathcal{T}_Q$ that are not involved in the intersection [3]. If this is true, the intersection point lies on an edge of $\mathcal{T}_P$ and is inside $\mathcal{T}_Q$ and the *intersection invariant* holds, see the yellow dots in Figure 3.2. At this point, we need to remember, which supporting line of $\mathcal{T}_Q$ contains the point, i.e. which of e_0, e_1 or e_2 was used to determine the intersection. Depending of if there exist intersection points or not, different steps are necessary.

Intersection points exist. If one or more points are found that occur on an edge of $\mathcal{T}_P$ and are inside $\mathcal{T}_Q$, there are three cases that allow for early termination: An edge of $\mathcal{T}_P$ and an edge of Q can intersect, intersections occur before and after an edge of $\mathcal{T}_Q$, or intersection can occur between $\mathcal{T}_P$ and two or more disjoint regions of $\mathcal{T}_Q$. If an intersection point is found to occur on both e_i and t_j, we have found an intersection between P and Q and the algorithm terminates [4].

For every intersection between e_i and t_j, that occurs on t_j, we remember if the intersection occurs before or after e_i. If at least one intersection occurs before e_i and at least one intersection occurs after e_i, then the line segment that connects those two intersections contains e_i and is contained in $\mathcal{T}_P$. Then we have found an intersection between Q and P and the algorithm terminates [5]. See Figure 3.3(a).

To determine if two different disjoint regions of $\mathcal{T}_Q \setminus \textsc{ch}(e_0, e_1, e_2)$ intersect with $\mathcal{T}_P$ we maintain three counters: r_0, r_1 and r_2. For every intersection between e_i and t_j, that occurs on t_j and is inside $\mathcal{T}_Q$, test if the intersection

[2] Line 99-113 in `IntersectBarpa.cpp`
[3] Line 150-178, 198-218 in `IntersectBarpa.cpp`
[4] Line 170-178 in `IntersectBarpa.cpp`
[5] Line 181-194 in `IntersectBarpa.cpp`

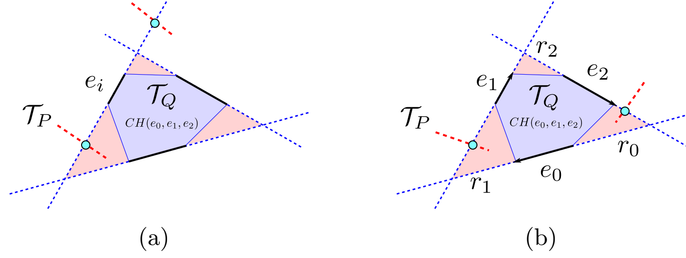


Figure 3.3: Two cases where the algorithm can find intersection and terminate. Figure 3.3(a) shows two intersections between $\mathcal{T}_P$ and $\mathcal{T}_Q$ that occur before and after the edge e_i. In this case, the line segment that connects the two intersection points is inside P and contains $e_i \subseteq Q$, thus P and Q intersect. In Figure 3.3(b) the relation between the edges e_0, e_1, and e_2 and the disjoint regions r_0, r_1, and r_2 is illustrated. The edges that define $\mathcal{T}_Q$ have their directions indicated, such that the two cyan intersection points occur before e_1 and after e_2, thus $\mathcal{T}_P$ intersects r_0 and r_1 and P and Q intersect.

occurs before or after e_i. If the intersection occurs before e_i, increment r_i, otherwise increment $r_{i-1 \bmod 3}$, i.e. we count how many intersections occur in each of the disjoint regions $\mathcal{T}_Q \setminus \text{CH}(e_0, e_1, e_2)$. If more than one of r_0, r_1, and r_2 is greater than zero, i.e. if there is an intersection in two different disjoint regions, there exist a line segment in $\mathcal{T}_P \subseteq P$ that connects two points such that the line segment intersects $\text{CH}(e_0, e_1, e_2) \subseteq Q$. Then the algorithm has found an intersection between Q and P and terminates[6]. How the disjoint regions are indexed in relation to the edges of $\mathcal{T}_Q$ is illustrated in Figure 3.3(b).

If intersection points exist and the algorithm has not terminated, the intersection subroutine is used.

No intersection points exist. If no intersection point is found to lie inside or on the boundary of $\mathcal{T}_Q$ and occur on an edge of $\mathcal{T}_P$, then the boundaries of $\mathcal{T}_Q$ and $\mathcal{T}_P$ do not intersect. The shapes may still intersect, thus we have to test if $\mathcal{T}_P$ is completely contained in $\mathcal{T}_Q$ or vice versa. This is done by testing if v_0 is inside $\mathcal{T}_Q$ and w_0 is inside $\mathcal{T}_P$. If w_0 is inside $\mathcal{T}_P$, a point in Q is contained in P and the two intersect. If v_0 is inside $\mathcal{T}_Q$, we have that Invariant 2 is maintained and the intersection invariant subroutine is used [7].

Otherwise, Invariant 2 can not hold, i.e. $\mathcal{T}_P$ and $\mathcal{T}_Q$ are not inside each other and none of their edges intersect, and Invariant 1 must hold. In order to use the separation invariant subroutine, we have to find a separating line that is tangent to one of v_i.

Searching for a separating line. We show in the following lemma that there exists a separating line that is parallel to an edge of $\mathcal{T}_P$ or $\mathcal{T}_Q$. This means there exist only six candidates for the angle of the separating line, and three vertices the line can be tangent to.

[6] Line 210-215, 221-232 in `IntersectBarpa.cpp`
[7] Line 263-278 in `IntersectBarpa.cpp`

Lemma 1. *Given two separable convex polygons P and Q, there exists at least one separating line that is parallel to an edge of P or Q.*

Proof. Consider the two convex polygons P and Q. Assume there exists a line that separates P and Q. Let the origin of the coordinate system be at the center of Q. Let tQ be the polygon where every vertex of Q is multiplied by the scalar $t > 1$, i.e. the polygon Q that is scaled uniformly by factor t. Uniform scaling maintains the angles of all edges, so every edge of tQ is parallel to an edge of Q. Consider tQ while increasing t continuously. The shapes tQ and P must intersect for some value of t, because when $t = \infty$, tQ spans the whole space and contains P. Let t_c be the smallest $t > 1$ such that the boundaries of P and t_cQ intersect. There are three possible ways for P and t_cQ intersect. Either two parallel edges intersect, a vertex intersects an edge, or two vertices intersect.

- If two parallel edges intersect, then both edges support a separating line by convexity of P and $Q \subset t_cQ$.
- If an edge and a vertex intersect, then the edge supports a separating line by convexity of P and $Q \subset t_cQ$.
- If two vertices intersect, consider the two edges on each polygon that they border. Two consecutive edges on a polygon form a corner. Each corner has some angle inside the shape it belongs to. The angles for the two corners can either be equal or different. Let the angles be α and β such that $\alpha \geq \beta$.
  - Assume $\alpha > \beta$. Consider the the lines that extend the edges of α. If none of them intersect the other shape, at least one of them is a separating line. If one of them intersects, the other must be a separating line: Assume for contradiction that both lines intersect the other shape. Then β must be at least as large α, which is a contradiction.
  - Assume that $\alpha = \beta$. Consider the lines that extend the edges α. The other shape can now only intersect one of the lines or its edges can be contained in the lines. If the shape intersects one of the lines, the other line is a separating line. If the edges are contained in a line each, both lines are separating lines.

We have found an edge that is parallel to a separating line for each case, because either an edge of P supports a separating line, or an edge of t_cQ supports a separating line and all of t_cQ's edges are parallel to edges of Q. □

Finding the separating line in constant time. Lemma 1 tells us that we only have to test the six lines that extend the six edges of $\mathcal{T}_P$ and $\mathcal{T}_Q$. We start by testing the edges of $\mathcal{T}_Q$ and then we test the edges of $\mathcal{T}_P$. It is easy to determine if an edge e of $\mathcal{T}_Q$ can be used as a separating line. Because $\mathcal{T}_P$ is a

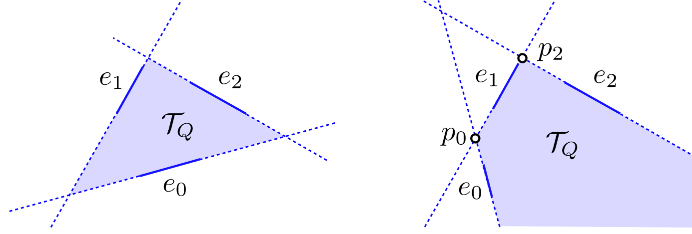


Figure 3.4: The two possible shapes of $\mathcal{T}_Q$. In 3.4(a) $\mathcal{T}_Q$ is a triangle which is the edge hull of e_0, e_1 and e_2. In 3.4(b) $\mathcal{T}_Q$ is a convex semi infinite region, because the edges are arranged differently. The points p_0 and p_1 are the corners of the semi infinite regions, which can be used as origins for the rays that bound $\mathcal{T}_Q$. Note that if $\mathcal{T}_Q$ is a convex semi infinite region, e_1 is not necessarily the 'middle' edge that supports an edge and not a ray that shapes $\mathcal{T}_Q$. In the first iteration of the algorithm, any edge could be the middle edge. Only in the following iterations, we know that e_1 is the middle edge like in Figure 3.4(b), because edges are labelled accordingly.

triangle, if the vertices v_0, v_1, and v_2 are to the left of e, then $\mathcal{T}_P$ must be to the left of e, in which case e supports a separating line [8].

Testing if an edge t of $\mathcal{T}_P$ separates $\mathcal{T}_P$ and $\mathcal{T}_Q$ is more difficult, especially when $\mathcal{T}_Q$ is a convex semi infinite region. To determine if $\mathcal{T}_Q$ is to the left of t, we first have to determine if the edge hull $\mathcal{T}_Q$ is a triangle or a semi infinite convex region, see Figure 3.4.

Determining the shape of $\mathcal{T}_Q$. We can determine if $\mathcal{T}_Q$ is semi infinite or a triangle in the following way. If all three intersection points of the lines supported by the edges of $\mathcal{T}_Q$ border $\mathcal{T}_Q$, i.e. are to the right of or on all three edges, then $\mathcal{T}_Q$ is a triangle and the intersection points are its vertices. Otherwise, one of the intersection points lies to the left of one edge, in which case $\mathcal{T}_Q$ is a semi infinite convex region [9].

$\mathcal{T}_Q$ is a triangle. If $\mathcal{T}_Q$ is a triangle, we can find the three corners of the triangle by determining the intersection between each edge of $\mathcal{T}_Q$, which we already have done to determine the shape of $\mathcal{T}_Q$. If all three corners are to the left of t, then $\mathcal{T}_Q$ is to the left of t, in which case t supports a separating line [10].

$\mathcal{T}_Q$ is semi infinite. If $\mathcal{T}_Q$ is a convex semi infinite region, we have to determine if its two corners are to the left of t, and if the two rays that extend from the corners do intersect with the line supported by t. Let e_0 and e_2 support the lines whose intersection point lies outside $\mathcal{T}_Q$. Let p_0 the intersection point between the lines that e_0 and e_1 support and p_2 the intersection point between the lines that e_1 and e_2 support, see Figure 3.4(b). We use the directed line segments (p_0, w_0) and (p_2, w_2) to represent the two rays. Compute the two

[8] Line 512-562 in `IntersectBarpa.cpp`
[9] Line 404-472 in `IntersectBarpa.cpp`
[10] Line 429-451 in `IntersectBarpa.cpp`

intersections between the rays and the line supported by t. If both intersections occur before the two rays or there is no intersection because the objects are parallel, and p_0 and p_2 are to the left of t, then the line supporting t does not intersect $\mathcal{T}_Q$ and it separates $\mathcal{T}_P$ and $\mathcal{T}_Q$ [11].

Finding a vertex that the separating line is tangent to. Having found a separating line, we now establish Invariant 1. To establish the invariant, we have to find a vertex of $\mathcal{T}_P$ that is tangent to a separating line. If the separating line is supported by an edge of $\mathcal{T}_P$, we can use any vertex that is part of this edge. Otherwise, the separating line is supported by an edge of $\mathcal{T}_Q$. In this case, we find the vertex that is closest to the separating line, and construct a line parallel to the separating line tangent to that vertex [12]. This approach will yield a vertex that is tangent to a separating line according to Lemma 2. Having found a vertex that is tangent to a separating line, we use the separation invariant subroutine.

Lemma 2. *Let two convex polygons P and Q be separated by a line ℓ. There exists a parallel line that separates P and Q, that is tangent to at least one vertex of P.*

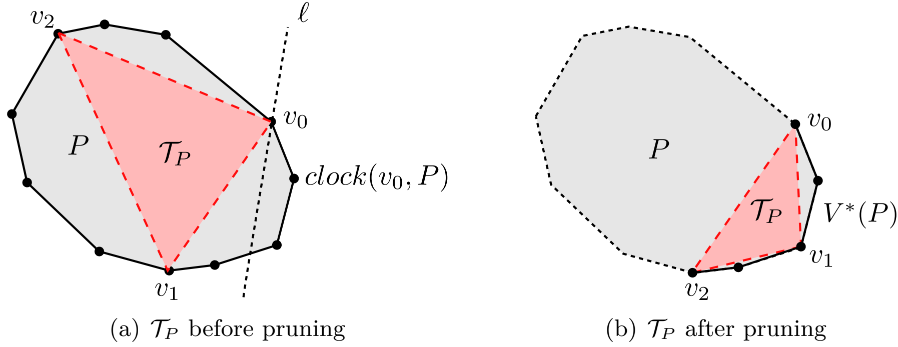
Proof. For simplicity, we assume that P is to the left of ℓ, but a symmetric argument can be made for the opposite case. Let p be be a point on the boundary of P, such that the distance between p and l is minimal. Let ℓ' be the line that is parallel to ℓ and tangent to p. Assume for contradiction that there exists a point p' that is to the right of ℓ' and inside P. If p' is to the left of or on ℓ, the distance between p and l is not minimal and we have a contradiction. If p' is to the right of ℓ, it is not a separating line, and we have a contradiction. Thus, P is to the left of ℓ'. Per construction, Q is to the right of ℓ', because Q is already to the right of ℓ. Thus ℓ' is a separating line.

The point p can either be a vertex or a point on an edge of P. If p is a vertex, the separating line ℓ' is tangent to it. If p is a point on an edge of P, the edge is parallel to ℓ' and the separating line is also tangent to the vertices that define the edge. Assume for contradiction that the edge is not parallel to ℓ' and that the point is not a vertex. Because ℓ' is a separating line, the vertices of the edge have to lie to the left of ℓ' and one of the endpoints of the edge is the closest point to ℓ because the edge is not parallel to ℓ. Because p is not a vertex, it can not be the closest point to ℓ and we have a contradiction, thus the edge is parallel to ℓ or p is a vertex. □

3.1.6 Separation Invariant Subroutine

If Invariant 1 holds, we have a separating line ℓ and a vertex v_i, where $i \in \{0, 1, 2\}$, on $\mathcal{T}_P$, such that ℓ is tangent to v_i. We want to find a part of P that can intersect with Q and prune everything that can not intersect Q. Consider the two neighbours of v_i on the boundary of P, $clock(v_i, P)$ and $counter(v_i, P)$.

[11] Line 454-508 in `IntersectBarpa.cpp`
[12] Line 552-562 in `IntersectBarpa.cpp`



(a) $\mathcal{T}_P$ before pruning

(b) $\mathcal{T}_P$ after pruning

Figure 3.5: In 3.5(a), the line ℓ separates $\mathcal{T}_P$ from $\mathcal{T}_Q$. The separating line is tangent to v_0. Because the clockwise neighbour to v_0 is on the opposite side of ℓ, the algorithm keeps only the indicies in $chain(v_0, v_1, P)$, while the rest of the vertices is pruned. Figure 3.5(b) illustrates $\mathcal{T}_P$ after the dashed chain is pruned and the indices v_0, v_1 , and v_2 rearranged such that they are in clockwise order and pruned elements can be identified easily.

Note that v_0 and v_2 are neighbours, if P has been pruned before, see Figure 3.5(b). Now one of the neighbours can be on the other side of ℓ as the rest of $\mathcal{T}_P$, or both neighbours can be on the same side as $\mathcal{T}_P$. We describe both cases in the following paragraphs.

Separation. If both neighbours are on the same side of ℓ as $\mathcal{T}_P$, then by the convexity of P, all of P must lie to the same side of ℓ, in which case P and $Q \subseteq \mathcal{T}_Q$ are on opposite sides of ℓ and they do not intersect and the algorithm terminates [13].

Pruning. If P and Q are not separate, only one of v_i's neighbours can lie on the same side as Q, otherwise P would not be convex. If the clockwise neighbour is on the same side as Q, like in Figure 3.5(a), an intersection between P and Q can only occur on the polygonal chain $C = chain(v_i, clock(v_i, \mathcal{T}_P), P)$. Set $v_0 = v_i$, $v_2 = clock(v_i, \mathcal{T}_P,)$ and set v_1 to be the median index of C.

If the counter clockwise neighbour of v_i is on the same side as Q, an intersection between P and Q can only occur on the boundary on the polygonal chain $C = chain(counter(v_i, \mathcal{T}_P), v_i, P)$. In this case, set $v_0 = counter(v_i, \mathcal{T}_P)$, $v_2 = v_i$, and set v_1 to be the median index of C. In any case, we now have that $V^*(P) = C$.

Doing this, we have pruned a fraction of $V^*(P)$ and the remaining part is $chain(v_0, v_2, P)$ [14]. Note that rearranging the indices of v_0 to v_2 causes v_1 to be clockwise to v_0 and v_2 clockwise to v_1. This allows us to easily identify, which indices have been pruned. All pruned index will be contained in $chain(clock(v_2, P), counter(v_0, P), P)$. Figure 3.5(b) illustrates $\mathcal{T}_P$ after being pruned. Because $\mathcal{T}_P$ has changed, we have to again determine which invariant holds and use the according subroutine.

[13] Line 575-624 in `IntersectBarpa.cpp`
[14] Line 626-676 in `IntersectBarpa.cpp`

Correctness. We argue that that the separation invariant subroutine maintains Invariant 3 in the following lemma.

Lemma 3. *Using the separation invariant subroutine maintains Invariant 3.*

Proof. The separation invariant subroutine is only used if Invariant 1 holds. Then there is a line ℓ than separates $\mathcal{T}_Q$ and $\mathcal{T}_P$. The line is also tangent to some vertex v of $\mathcal{T}_P$. Let ℓ^- be the halfplane supported by ℓ that contains $\mathcal{T}_P$ and let ℓ^+ be the halfplane that contains $\mathcal{T}_Q$. We can split P into two parts; $P \cap \ell^+$ and $P \cap \ell^-$. Assume that P intersect Q. Because ℓ separates $\mathcal{T}_P$ from $Q \subseteq \mathcal{T}_Q$, then $P \cap \ell^-$ can not intersect with Q, which means that $P \cap \ell^+$ must intersect Q. Thus, P and Q intersect only if $P \cap \ell^+$ intersects Q. Trivially, if $P \cap \ell^+$ intersects Q, then P intersects Q.

Now we have to ensure, that $V^*(P)$ contains $P \cap \ell^+$. Consider the two neighbours $\mathit{clock}(v, P)$ and $\mathit{counter}(v, P)$. If both are inside ℓ^-, then by convexity of P, $P \cap \ell^+$ is empty. Otherwise only one of them can be in ℓ^+, because P is convex. We can split the boundary of P in to three polygonal chains, one between v_0 and v_1, one between v_1 and v_2, and one between v_2 and v_0. One of those three vertices is v. Now only one of those polygonal chains can intersect with ℓ^+ (otherwise we could have that both neighbours of v can be in ℓ^+). By choosing $V^*(P)$ to be the chain that intersects ℓ^+, we have that $(P \cap \ell^+) \subseteq V^*(P)$. This is because the first endpoint of the chain is v, and the other endpoint is inside ℓ^-, while some elements on the chain are in ℓ^+, we have that all elements in ℓ^+ are on the chain (otherwise there would be a hole in the chain). □

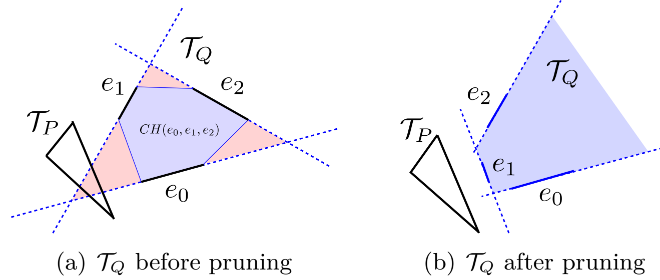
3.1.7 Intersection Invariant Subroutine

The intersection invariant subroutine is only used if Invariant 2 applies. The edge hull $\mathcal{T}_Q$ can be split in to three disjoint regions $\mathcal{T}_Q \backslash \text{CH}(e_0, e_1, e_2)$, as seen in Figure 3.6(a). From determining which invariant holds, we have that either $\mathcal{T}_P$ intersects the boundary of a disjoint region, or found a vertex of $\mathcal{T}_P$ that is inside a disjoint region. We describe how to deal with both cases in the following two paragraphs.

Boundaries intersect. If we have found an intersection between lines, the intersection point can either be before or after an edge e_i, where $i \in \{0, 1, 2\}$, that is involved in the intersection. If the intersection has occurred on e_i, the algorithm has terminated and found an intersection. If the intersection point occurs after e_i, then the edges e_i and $\mathit{clock}(e_i, \mathcal{T}_Q)$ bound the region which $\mathcal{T}_P$ intersects. To prune Q, set $e_0 = e_i$ and $e_2 = \mathit{clock}(e_i, \mathcal{T}_Q)$. The case where the intersection occurs after e_0 is illustrated in Figure 3.6. Otherwise, the intersection occurs before e_i. Then e_i and $\mathit{counter}(e_i, \mathcal{T}_Q)$ bound the region. To prune Q, set $e_0 = \mathit{counter}(e_i, \mathcal{T}_Q)$ and $e_2 = e_i$ [15].

In any case, set e_1 to be the median edge on $C = \mathit{chain}(e_0.\mathit{start}, e_2.\mathit{end}, Q)$, after e_0 and e_2 are reassigned. The remaining vertices are $V^*(Q) = C$ and $E^*(Q)$ is the set of edges on C, but not the edge connecting $e_0.\mathit{start}$ and $e_2.\mathit{end}$.

[15] Line 239-257, 338-361 in `IntersectBarba.cpp`



(a) $\mathcal{T}_Q$ before pruning (b) $\mathcal{T}_Q$ after pruning

Figure 3.6: If $\mathcal{T}_P$ intersects exactly one disjoint region $\mathcal{T}_Q \setminus \text{CH}(e_0, e_1, e_2)$, like in Figure 3.6(a) where the intersection between $\mathcal{T}_P$ and $\mathcal{T}_Q$ occurs after e_0 and before e_1, then the result is that the edge e_1 is relabelled to e_2 and a new e_1 is chosen to be the median edge on $chain(e_0.end, e_2.start, Q)$ in Figure 3.6(b).

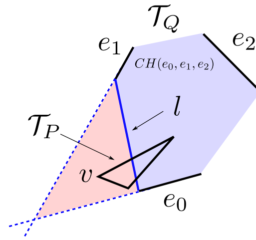


Figure 3.7: If $\mathcal{T}_P$ is contained in $\mathcal{T}_Q$, and find some vertex v of $\mathcal{T}_P$ that is to the left of an edge l of $\text{CH}(e_0, e_1, e_2)$. If any other vertex is to the right of l, the line segment between those two vertices must intersect $\text{CH}(e_0, e_1, e_2)$.

Vertex inside edge hull. If we have found that some vertex v of $\mathcal{T}_P$ inside $\mathcal{T}_Q$. We can determine in which of the three regions v is and if $\mathcal{T}_P$ intersects $\text{CH}(e_0, e_1, e_2)$. Consider the directed line segments that for each $i \in \{0, 1, 2\}$ connect $e_i.end$ to $clock(e_i, \mathcal{T}_Q).start$. These are illustrated as the thin blue lines in Figure 3.6(a). If v is to the right of those three line segments, v is inside $\text{CH}(e_0, e_1, e_2) \subseteq Q$ and the algorithm terminates[16]. Otherwise, v must be to the left of exactly one line segment. Let this line segment be $l = (e_i.end, clock(e_i, \mathcal{T}_Q).start)$.

Before we prune Q accordingly, we test if $\mathcal{T}_P$ intersects $\text{CH}(e_0, e_1, e_2)$, by computing the sidedness of the other two vertices of $\mathcal{T}_P$ of l. If they do not both have the same sidedness as v, $\mathcal{T}_P$ intersects $\text{CH}(e_0, e_1, e_2)$ and the algorithm terminates[17]. This situation is illustrated in Figure 3.7.

To prune Q, set $e_0 = e_i$ and $e_2 = clock(e_i, \mathcal{T}_Q)$ and set e_1 to be the median edge on the boundary $C = chain(e_0.start, e_2.end, Q)$[18]. The remaining vertices are $V^*(Q) = C$ and $E^*(Q)$ is the set of edges on C, but not the edge connecting $e_0.start$ and $e_2.end$.

Correctness. When Invariant 2 holds and the intersection invariant subroutine is called, we have either found an intersection between the boundaries of

[16]Line 281-301 in `IntersectBarba.cpp`
[17]Line 312-328 in `IntersectBarba.cpp`
[18]Line 338-361 in `IntersectBarba.cpp`

$\mathcal{T}_P$ and $\mathcal{T}_Q$, found a vertex of $\mathcal{T}_P$ inside $\mathcal{T}_Q$, or found a vertex of $\mathcal{T}_Q$ inside $\mathcal{T}_P$. We can ignore the case where a vertex of $\mathcal{T}_Q$ is inside $\mathcal{T}_P$, because then the algorithm has terminated correctly already. The two remaining cases are that the boundaries of $\mathcal{T}_P$ and $\mathcal{T}_Q$ intersect, or that $\mathcal{T}_P$ is inside $\mathcal{T}_Q$. We know that if $\mathcal{T}_P$ intersects $\textsc{ch}(e_0, e_1, e_2)$ that P and Q intersect. Thus we have to consider the following cases:

- $\mathcal{T}_P$ is inside $\mathcal{T}_Q$
  - $\mathcal{T}_P$ intersects $\textsc{ch}(e_0, e_1, e_2)$.
  - $\mathcal{T}_P$ is inside one disjoint region.
- The boundary of $\mathcal{T}_P$ and $\mathcal{T}_Q$ intersect.
  - $\mathcal{T}_P$ intersects $\textsc{ch}(e_0, e_1, e_2)$.
  - $\mathcal{T}_P$ intersects the boundary of one disjoint region.

We start with considering that a vertex v of $\mathcal{T}_P$ is inside $\mathcal{T}_P$ and their boundaries do not intersect. If the vertex is inside $\textsc{ch}(e_0, e_1, e_2) \subseteq Q$, then P and Q must intersect. Otherwise the vertex is outside $\textsc{ch}(e_0, e_1, e_2)$ but inside $\mathcal{T}_Q$, which means it must be in one of the disjoint regions. If one of the other two vertices of $\mathcal{T}_P$ is not in the same disjoint region, then it is either inside $\textsc{ch}(e_0, e_1, e_2)$ or in another disjoint region. If it is inside in $\textsc{ch}(e_0, e_1, e_2)$, we have intersection between P and Q. If it is in another disjoint region, then we can construct a line segment between it and v, which is inside $\mathcal{T}_P$ and intersects $\textsc{ch}(e_0, e_1, e_2)$ and we again have an intersection between P and Q. It remains only that $\mathcal{T}_P$ is inside exactly one disjoint region.

Now consider the case where the boundaries of $\mathcal{T}_P$ and $\mathcal{T}_Q$ intersect. Now $\mathcal{T}_P$ can intersect $\textsc{ch}(e_0, e_1, e_2)$ or the boundary of exactly one disjoint region. We know that $\mathcal{T}_P$ intersects $\textsc{ch}(e_0, e_1, e_2)$ if one of the following is true:

- An edge of $\mathcal{T}_P$ intersects one of e_0, e_1, or e_2.
- $\mathcal{T}_P$ contains e_0, e_1, or e_2.
- $\mathcal{T}_P$ intersects with the boundaries of more than one disjoint region.

The first two cases on this list immediately tell us that P and Q intersect. In the third case, the line segments that connects two intersection points on the boundary of two different disjoint regions is inside $\mathcal{T}_P$ and intersects $\textsc{ch}(e_0, e_1, e_2)$, thus P and Q intersect. Note that this list is not exhaustive, because the boundary of $\mathcal{T}_P$ might intersect the boundary of one disjoint regions and also intersect $\textsc{ch}(e_0, e_1, e_2)$. This is not of great importance, as long as we maintain Invariant 3. It only remains that $\mathcal{T}_P$ intersects the boundary of one disjoint region.

We now have exactly two cases to consider and both cases involve the intersection between $\mathcal{T}_P$ and one disjoint region of $\mathcal{T}_Q$. Either $\mathcal{T}_P$ intersects the boundary of the disjoint region, or $\mathcal{T}_P$ is completely inside the disjoint region. We first consider the case where $\mathcal{T}_P$ intersects the boundary of the disjoint region. In this case, it is not possible that Q contains P, because we have

that some part of P is outside $\mathcal{T}_Q$, otherwise $\mathcal{T}_P$ could not cross and intersect a boundary of $\mathcal{T}_Q$. This means that P and Q intersects if and only if their boundaries intersect or Q is inside P. Barba et al. proof that in this case, P and Q intersect if and only if P and $\textsc{ch}(E^*(Q))$ intersect [1], by showing that there must exists an intersection point on the remaining boundary of $\textsc{ch}(E^*(Q))$ if P and Q intersect.

It remains to proof that if $\mathcal{T}_P$ is contained in $\mathcal{T}_Q$, then P and Q intersect if and only if $E^*(Q)$ intersects with P. We show this in the following lemma.

Lemma 4. *The intersection invariant subroutine maintains Invariant 3 if $\mathcal{T}_P$ is inside a disjoint region of $\mathcal{T}_Q$.*

Proof. If $\mathcal{T}_P$ is inside a disjoint region, it does not intersect with the boundaries of the disjoint regions. Thus we have that there exists a line ℓ that separates $\mathcal{T}_P$ and $\textsc{ch}(e_0, e_1, e_2)$ and is tangent to an edge of $\textsc{ch}(e_0, e_1, e_2)$, see Figure 3.8(a). Let ℓ^+ be the halfplane supported by ℓ that contains $\textsc{ch}(e_0, e_1, e_2)$ and let ℓ^- be its complement, i.e. the halfplane that contains $\mathcal{T}_P$. Assume that P and Q intersect. Now it must hold that $P \cap Q \cap \ell^- \neq \emptyset$ or $P \cap Q \cap \ell^+ \neq \emptyset$. If $P \cap Q \cap \ell^- \neq \emptyset$, then any elements of $Q \cap \ell^+$ are not necessary to find an intersection between P and Q.

Otherwise, if $P \cap Q \cap \ell^+ \neq \emptyset$, then a point that is in P must also be in $\mathcal{T}_Q$, because $Q \subseteq \mathcal{T}_Q$. Then there exists a point p_1 from P in $\ell^+ \cap \mathcal{T}_Q$ and a point p_2 from P that is in $\mathcal{T}_P \subset \ell^-$. Consider the line segment (p_1, p_2). Because this line segment is contained in $\mathcal{T}_Q$ and intersects ℓ, it must intersect the boundary of $\textsc{ch}(e_0, e_1, e_2)$ which ℓ is tangent to, see Figure 3.8(b). This intersection between (p_1, p_2) and ℓ is inside both ℓ^+ and ℓ^-. Thus if P and Q intersect, then P must intersect $Q \cap \ell^-$ and we have that P and Q intersect only if P intersects $Q \cap \ell^-$. It trivially holds that if P intersects $Q \cap \ell^-$, then P and Q intersects. Thus we have shown that P and Q intersect if and only if P intersects $Q \cap \ell^-$.

It remains to show that $Q \cap \ell^- \subseteq E^*(Q)$. The algorithm always keeps the two edges that bound the disjoint region. It only discards elements on the polygonal chain between those two edges that contains the third edge of $\mathcal{T}_Q$, which is not a neighbour to the disjoint region, e.g. edge e_2 in Figure 3.8(a). This polygonal chain is completely contained in ℓ^+, so no elements of Q in ℓ^- are pruned. Thus $Q \cap \ell^- \subseteq E^*(Q)$. □

3.1.8 Constant Size

When both polygons are pruned to three vertices or edges respectively, the correctness invariant states, that P and Q intersect if and only if the remaining shapes intersect. To determine if the remaining shapes intersect, we consider $\textsc{ch}(e_0, e_1, e_2)$. Because there are only three edges remaining in $E(Q^*)$, these three edges are neighbours on the boundary of Q and then $\textsc{ch}(e_0, e_1, e_2)$ has only four edges. The fourth edge is $e_3 = (e_0.start, e_2.end)$. Compute the intersection between the edges e_0 to e_3 and the edges t_0 to t_2. If an intersection occurs on an edge of P and an edge of $\textsc{ch}(e_0, e_1, e_2)$, we have found an intersection between P and Q and the algorithm terminates. If no such intersection is found, we test

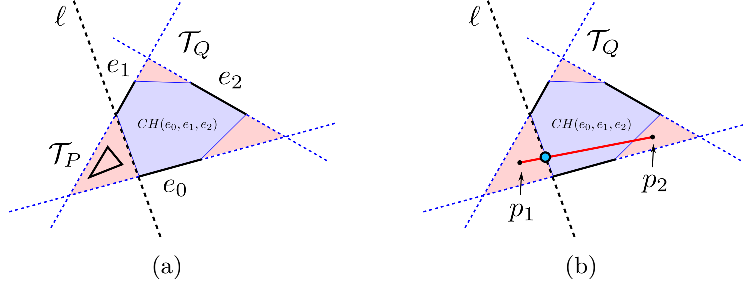


Figure 3.8: In 3.8(a), we have that $\mathcal{T}_P$ is completely contained inside one disjoint region of $\mathcal{T}_Q$, thus there exists a separating line ℓ between $\mathcal{T}_P$ and $\text{CH}(e_0, e_1, e_2)$. If any intersection between P and Q occurs to the right of ℓ in Figure 3.8(b), then we can trace a line segment from $\mathcal{T}_P$ to that point and the line segment will intersect $\text{CH}(e_0, e_1, e_2)$.

if v_0 is inside all four edges, in which case P intersects Q. Otherwise, we test if w_0 is inside all three edges of $\mathcal{T}_P$, in which case Q intersects P. If none of these cases occurs, the remaining shapes do not intersect, thus P and Q do not intersect either [19].

3.1.9 Running time and Correctness

Running Time. Barba et al. state that this algorithm prunes a constant fraction of either $V^*(P)$ or $E^*(Q)$ in each iteration [1, Theorem 2.1], thus after after $O(\log n)$ iterations only a constant number of elements remain, where $n = size(P) + size(Q)$. We then use constant time to determine if the remaining elements intersect or not, and the algorithm terminates.

Correctness of the complete algorithm. If the algorithm does not find a separating line or an intersection between P and Q, then after $O(\log n)$ iterations, $V^*(P)$ and $E^*(Q)$ are of constant size. Because of Invariant 3 we know that P and Q intersect if and only if $\text{CH}(V^*(P))$ and $\text{CH}(E^*(Q))$ intersect. Because $\text{CH}(V^*(P))$ is equal to $\mathcal{T}_P$ and $\text{CH}(E^*(Q))$ is equal to the polygon consisting of e_0, e_1, e_2, and e_3, we determine if these two convex hulls intersect with the subroutine in Section 3.1.8 and the algorithm terminates correctly.

3.1.10 Implementation Details

In this section, we describe additional implementation details, that are not specific to either subroutine.

Intersections. At the start of the algorithm, nine intersection have to be computed. To compute an intersection between two line segments that are represented in a parametrized fashion is costly, this implementation uses up to eight multiplications, two division, and several additions and subtractions with double precision to compute an intersection. To reduce the number of

[19] Line 680-740 in `IntersectBarba.cpp`

intersection computations, we use the fact that not all edges of $\mathcal{T}_P$ and $\mathcal{T}_Q$ are changed at the end of each iteration.

When the separation invariant subroutine is used, only one of the three vertices of $\mathcal{T}_P$ is changed, which results in two new edges. The third edge is the same as before, so we reuse the three intersection calculations that involve that edge and only do six intersection computations for the two new edges [20].

When the intersection invariant subroutine is used, only one of the three edges of $\mathcal{T}_Q$ changes. Thus we can reuse six of the intersection computations that involve the two old edges and compute three new intersections involving the new edge [21].

Finding a separating line. When searching for a separating line to use in the separation invariant subroutine, it matters what edges are considered first. To test if an edge of $\mathcal{T}_Q$ separates $\mathcal{T}_Q$ and $\mathcal{T}_P$ uses only three sidedness tests. To compute if an edge of $\mathcal{T}_P$ separates the two is much more expensive, because the algorithm needs to compute three intersections to determine if $\mathcal{T}_Q$ is a triangle or an convex semi-infinite region, in which case more intersection computations are necessary to determine if an edge of $\mathcal{T}_P$ crosses the rays that border $\mathcal{T}_Q$. It might therefore be best to first consider the edges of $\mathcal{T}_Q$ and try to avoid the more expensive calculations.

On the other hand, if we first search through the edges of $\mathcal{T}_P$ and find a separating line, then this line is parallel to an edge of $\mathcal{T}_P$. Then the neighbouring vertices on the boundary of P will not both be on the same side of the separating line, unless the edge is also an edge on the boundary of P. This means that it is more improbable that we can use early termination when P and Q are separate.

We have chosen to test the edges of $\mathcal{T}_Q$ first, so that it is more probable to find a separating line and terminate early, but it is not necessarily the most efficient solution.

Tangent edges and vertices. It is possible that two convex polygons are tangent, either by sharing a vertex or by having a vertex that lies on an edge of the other polygon. In these cases, we consider the polygons to intersect, because they both contain the same point and thus $P \cap Q \neq \emptyset$. In the implementation, this is achieved to some degree, by considering the endpoints of edges whenever the intersection between two edges is computed, i.e. when the intersection computation returns s and t, the edges intersect if $s, t \in [0, 1]$. Also, when using sidedness tests, a possible result is that a vertex lies on the line that extends some edge, instead of being to the left or right of it. This might happen when considering if some vertex is inside $\mathcal{T}_Q$ or not, in which case we consider the vertex to still be inside $\mathcal{T}_Q$.

These consideration are only of limited use, because computations involving floating point numbers tend to be imprecise and testing equality with floating point numbers is usually considered futile. We can however construct nice

[20] Line 655-670 in `IntersectBarba.cpp`

[21] Line 369-381 in `IntersectBarba.cpp`

inputs, where edges are axis aligned or vertices have integer coordinates, where these consideration do make a difference. For example two polygons that share an axis aligned edge, polygons that have partly overlapping axis aligned edges, and a polygons whose vertex lies on an axis aligned edge are correctly found to be intersecting by our implementation.

3.2 Dobkin and Souvaine's Algorithm

Given two convex polygons P and Q, we want to determine whether they intersect. The polygons are given as two arrays, with their vertices sorted in clockwise order.

The algorithm consists of three steps: First P and Q are split in to two polygons chains each, where P_L (resp. Q_L) is the left polygons chain of P (resp. Q) and P_R (resp. Q_R) is the right polygonal chain. Then P and Q intersect if and only if both pairs P_L and Q_R, and Q_L and P_R intersect. Finally do the following with both pairs. Prune both polygonal chains chains until one of them is of constant size. After one chain is pruned to constant size, prune the other chain of the pair to constant size. When both chains of a pair are of constant size, we can determine in constant time if they intersect.

In the following sections, we define P_L, P_R, Q_L, and Q_R. To construct these, the the top most and bottom vertices of P and Q are required. We present a method to find the top and bottom most vertices afterwards. We then present the algorithm that given a left chain L and a right chain R, prunes both chains until at least one of them is of constant size, using the results stated in Dobkin et al. [7]. Then we present an algorithm that prunes the rest of L or R until both are of constant size, and describe how to determine if they intersect. At the end of this section, we describe the complete algorithm and discuss implementation details. Our implementation of this algorithm can be found in `IntersectionDobkin2.cpp`.

3.2.1 Polygonal Chains

A closed convex polygon P has per definition vertices that are below or above all others, i.e. the polygon is finite. Let v_{hi} and v_{lo} be the highest and the lowest vertex of the polygon. We define two polygonal chains of P to be the chain of vertices $P_L = chain(v_{hi}, v_{lo}, P)$ and the chain of vertices $P_R = chain(v_{lo}, v_{hi}, P)$. We call P_L the left polygonal chain of P and P_R the right polygonal chain of P.

Semi infinite regions. In the rest of this thesis, we will consider P_L and P_R to be polygonal chains that define semi-infinite regions in the plane. These semi infinite regions consist of all the points that are to the right (using the sidedness definition from Section 2) of all edges on the chain and have y-coordinates smaller or equal to v_{hi} and larger or equal to v_{lo}. We use P_L and P_R to denote both the chains and the regions they define. This construction is visualized in

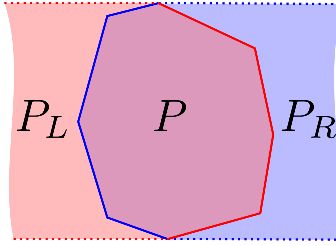


Figure 3.9: The regions P_L, P and P_R visualized. Note that $P = P_L \cap P_R$, thus $P \subseteq P_L$ and $P \subseteq P_R$. The blue edges are the polygonal chain defining P_R and the red edges are the polygonal chain defining P_L. The chain open to the left, P_L, is called the left polygonal chain, while P_R is the right polygonal chain.

Figure 3.9. Using this construction, polygon P is equal to the intersection of P_L and P_R.

Intersection of polygonal chains. Let L be the left polygonal chain of one polygon and R the right polygonal chain of another polygon. We say that L and R intersect if and only if the semi infinite regions they define intersect, i.e. $L \cap R \neq \emptyset$. We then use the following lemma from Dobkin et al.

Lemma 5. *"Let P and Q be convex polygons which have been divided into P_L, P_R, Q_L and Q_R as described above. Then P and Q intersect if and only if the polygonal pairs P_L and Q_R and P_R and Q_L intersect." [7, Lemma 1.]*

3.2.2 Binary Search for Extremum Points

In order to find the vertices necessary to construct the left and right polygonal chains of some convex polygon P, we have to find the top and bottom most vertices.

We define the topmost vertex v_t to be the vertex that has the largest y-coordinate of all vertices in P and whose counter clockwise neighbour is strictly below it, where t is the index of the top most vertex. Likewise, we define the bottom most vertex to be the vertex that has the smallest y-coordinate of all vertices in P and whose counter clockwise neighbour is strictly above it. These definitions ensure that if there are horizontal edges present in P, only one vertex can be the topmost one.

In the following, we describe how to determine the upward direction and a method to do binary search using this upward direction. We show that this method takes $O(\log n)$ time, where $n = size(P)$.

Determining upward direction. We want the upward direction of some vertex of a polygon to be clockwise or counter clockwise, such that when we traverse the boundary of the polygon in that direction, we encounter the topmost vertex before we encounter to bottom most vertex. To determine the upward direction of a vertex b, it is sufficient to look at the two neighbouring vertices. Let a, b and c be consecutive indices on the polygon, such that $b = clock(a, P)$ and $c = clock(b, P)$.

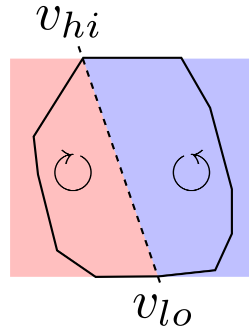


Figure 3.10: Upward direction some polygon. All vertices in the blue region have counter clockwise as their upward direction. All vertices in the red region have clockwise as their upward direction. The two vertices v_{hi} and v_{lo} are the top and bottom most vertices. The upward direction for v_{lo} is defined to be counter clockwise when searching for the top most vertex.

- If $b.y < c.y$, then t can be found by traversing the polygon in clockwise order.
- If $a.y > b.y$ then t can be found by traversing the polygon in counter clockwise order.
- If $a.y < b.y$ and $b.y \geq c.y$, we have found that b is the top most vertex.
- If $a.y == b.y$ have the same height, the upward direction is counter clockwise if none of the above apply, i.e. a would be the index of the topmost vertex. [22]

Using this definition of upward direction is illustrated in Figure 3.10.

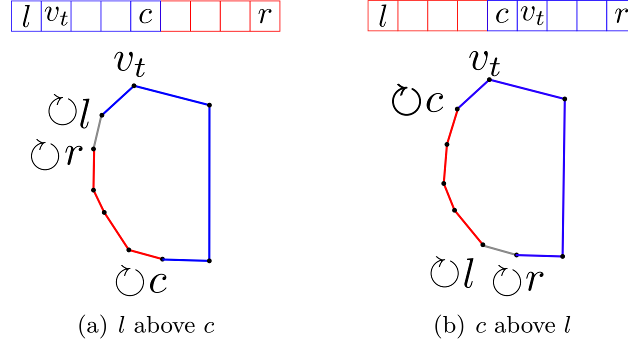
Search using upward direction. The algorithm is a binary search algorithm, that uses the upward direction to determine in which part of the array the topmost vertex can be found. It is initialised by storing two indices, $l = 0$ and $r = n - 1$, which initially point to the first and last position in the array the polygon is stored in. Let $direction(i)$ denote the upward direction of the vertex stored at index i in the array, as described in the above paragraph.

The algorithm loops while $l < r$. In each iteration, set $c = l + (r - l)/2$. If a vertex at index c, l, or r is the topmost vertex, the algorithm returns the index. Otherwise, there are four cases to consider. If $direction(c)$ is clockwise and $direction(r)$ is counter clockwise, set $l = c$ and continue the loop. If $direction(l)$ is clockwise and $direction(c)$ is counter clockwise, set $r = c$ and continue. If all three direction are clockwise, set $l = c$ if $c.y > l.y$, otherwise set $r = c$. If all three directions are counter clockwise, set $r = c$ if $c.y > r.y$, otherwise set $l = c$. [23]

Correctness

We have to argue that the top most vertex is not pruned during the algorithm, i.e. that $l \leq t \leq r$ is an invariant, and that the algorithm terminates. The

[22] Line 55-86 in `Common.cpp`
[23] Line 160-264 in `Common.cpp`



(a) l above c (b) c above l

Figure 3.11: These are the two possible cases that can occur, when all directions are clockwise during search for the top most vertex. The array that contains the vertices and the polygon it represents is shown. The blue edges and vertices are considered in the next iteration of the search, while red edges and vertices are pruned. When $c.y > l.y$, then the topmost vertex v_t lies between l and c, Figure 3.11(a). If $l.y > c.y$, then v_t lies between c and r, Figure 3.11(b).

following two lemmas cover the case where all directions are clockwise. We use these lemmas and their symmetric counterparts in the paragraph right after, to show that the rest of the cases are dealt with correctly. The following two lemmas are also illustrated in Figure 3.11

Lemma 6. *While searching for the topmost vertex in a convex polygon P. If the upward direction of indices $l < c < r$ is clockwise and $l.y > c.y$, then $l < t < c$.*

Proof. Let $l.y > c.y$, $l < c < r$, and $l < t < r$. Assume for contradiction that $c < t$. When traversing P in clockwise direction starting at l, each traversed vertex is above the last, until the top most vertex is reached. If $c < t$, vertex c is encountered before t while traversing the polygon, vertex v_c must be above v_l, which is a contradiction. Thus $v_l.y > v_c.y$ implies $t \leq c$. □

Lemma 7. *While searching for the topmost vertex in a convex polygon P. If the upward direction of indices $l < c < r$ is clockwise and $c.y > l.y$, then $c < t < r$.*

Proof. Let $c.y > l.y$, $l < c < r$, and $l < t < r$. Assume for contradiction that $t < c$. When traversing P in clockwise direction starting at l, each traversed vertex is above the last, until the top most vertex is reached. If $l < t < c$, then t is encountered before c because of the order of indices. Because $l.y < c.y \leq t.y$ and $direction(c)$ is clockwise, the traversal must encounter c before t, which is a contradiction. □

The topmost vertex is not pruned. When the algorithm is initialized, we have that $l \leq t \leq r$, because no vertices have been pruned and the topmost index must be stored in the array. We always choose c such that $l < c < r$. If $l = t$, $c = t$, or $r = t$, the algorithm terminates and has found the top most vertex. Now we must show that each iteration maintains that $l \leq t \leq r$ after

choosing $l = c$ or $r = c$. We have that $l < t < r$, so it must hold that either $l < t < c$ or $c < t < r$.

- Let $direction(l)$ be clockwise and $direction(c)$ be counter clockwise. If we would traverse the array starting from l in clockwise direction until we reach c, the upward direction would change somewhere in between. Because we start by traversing the array towards the top most vertex, the direction can only change at the topmost vertex or the bottom most vertex, and $l < c$, we have that $l < t < c$,
- Let $direction(c)$ be clockwise and $direction(r)$ be counter clockwise. By the same reasoning as above, we have that $c < t < r$.
- Let the direction of all three vertices be clockwise. If $l.y > c.y$, then $l < t < c$, according to Lemma 6. If $c.y > l.y$, then $c < t < r$, according to Lemma 7.
- Let the direction of all three vertices be counter clockwise. Let $c.y > r.y$. This is symmetric to the case in Lemma 7, because we could rotate the polygon 180 degrees and search for the bottom most vertex, which would be the exact same situation as in Lemma 7 but searching in the opposite direction. Thus $l < t < c$. Let $r.y > c.y$. This is symmetric to the case in Lemma 6, thus $c < t < r$.

It is not possible that two vertices have equal y-coordinates and have equal upward direction, because either one of them is the topmost vertex or one of them is the bottom most vertex, in which case their upward direction is different per our construction.

We have covered all possible cases that the algorithm can encounter and show that $l \leq t \leq r$ is maintained after reassigning l or r. If only a constant number of vertices is left, the topmost vertex must be among them.

The Algorithm Terminates. To show that the algorithm terminates, we must show that $l < r$ does not hold after a finite amount of iterations. Let l_i, c_i, and r_i denote l, r and c at the i'th iteration. At the start of the algorithm it holds that $l_0 < r_0 \Leftrightarrow r_0 - l_0 > 0$ and the algorithm terminates if $r_i - l_i \leq 0$. Let $d_i = r_i - l_i$. It holds that $r_{i+1} = c_i$ or $l_{i+1} = c_i$. We assume the former, but a symmetric argument can be made for the latter case.

Let $l_{i+1} = l_i$ and $r_{i+1} = \frac{r_i + l_i}{2}$, thus

$$\begin{aligned} d_{i+1} &= r_{i+1} - l_{i+1} \\ &= \frac{r_i + l_i}{2} - l_i \\ \Leftrightarrow 2d_{i+1} &= r_i - l_i = d_i \\ \Leftrightarrow d_{i+1} &= \frac{d_i}{2}, \end{aligned}$$

and the number of vertices is halved each iteration. Thus, the algorithm terminates after $O(\log_2 n)$ iterations, when only three or less vertices are left to consider.

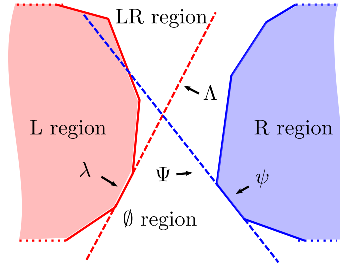


Figure 3.12: The lines Λ and Ψ support the edges λ and ψ. These edges are part of two different polygons. The edge λ is part of the left polygonal chain P_L of some polygon P, while the edge ψ is part of the right polygonal chain Q_R of some other polygon Q. The lines Λ and Ψ split the plane in to four regions. The L-region contains everything to the left of Λ and the R-region contains everything to the right of Ψ. The LR-region is the intersection of both regions, while the $\emptyset$-region is everything outside of these regions.

3.2.3 Binary Search on Polygonal Chains

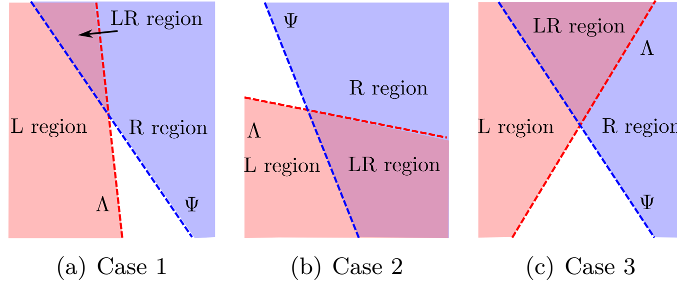
When the top most and bottom most vertices of P and Q are found, they are used to construct the four polygonal chains P_L, P_R, Q_L, and Q_R as described in Section 3.2.1. In this section, we are given either P_L and Q_R or Q_L and P_R as L and R. The aim is to find edges on L and R that are witness to an intersection between L and R. How to search on L and R is described in the two following sections.

Let L be a left polygonal chain and R be a right polygonal chain. Let λ denote an edge of L and let ψ denote an edge of R. Additionally, we use Λ and Ψ to denote the lines that extend λ and ψ. The lines Λ and Ψ can split the plane in up to four region. The L-region is everything to the left of Λ and the R-region is everything to the right of Ψ. Now L is contained in the L-region and R is contained in the R-region. The LR-region is the intersection of the L-region and the R-region. The $\emptyset$-region is the space not occupied by either. The LR-region is the space, in which vertices of both L and R could exist and intersections between L and R are possible. This construction is illustrated in Figure 3.12. The lines Λ and Ψ can be parallel, in which case we have to take some precautions. If the lines are not parallel, the LR-region can be above or below the $\emptyset$-region. How to deal with the parallel case and how to determine where the LR-region is located is described in the following paragraphs.

Parallel lines. If the lines Λ and Ψ are parallel, either the LR-region or the $\emptyset$-region can exist, but not both. If Ψ is to the left of Λ or the lines are equal, then the LR-region exists. Otherwise, the LR-region does not exist, and there can be no intersection between L and R [7, Footnote 2, Section 2.2]. [24]

Position of the LR-region. If the two lines are not parallel, it is possible to determine if the LR-region is above or below the $\emptyset$-region. Due to the way they are defined, these two regions can never lie side by side. Let $angle(\Psi)$ and $angle(\Lambda)$ denote the angle of Ψ and Λ, i.e. let $\Psi : y = ax+b$, then $angle(\Psi) = a$.

[24] Line 199-209 in `IntersectionDobkin2.cpp`



(a) Case 1 (b) Case 2 (c) Case 3

Figure 3.13: Example of the position of the LR-region. In this example, the angle of Ψ is below 0, and all possible configurations of Λ are shown. In 3.13(a), the angle of Λ is negative and smaller than the angle of Ψ. In 3.13(b), the angle of Λ is negative, but larger than the angle of Ψ. In 3.13(c), the angle of Λ is positive.

The lines Λ and Ψ can never be horizontal lines during the algorithm, because horizontal edges are discarded at the start of the algorithm. If one of Λ or Ψ is a vertical line, its slope is considered to be positive infinity, i.e. $angle(\Lambda) = \infty > 0$ or $angle(\Psi) = \infty > 0$ respectively. If $angle(\Psi) < 0$ there are the following three cases, which are illustrated in Figure 3.13.

1. If $angle(\Lambda) > 0$, then the LR-region is above.
2. If $angle(\Lambda) > angle(\Psi)$, then the LR-region is below.
3. Otherwise the LR-region is above.

If $angle(\Psi) > 0$ there are again three cases.

1. If $angle(\Lambda) < 0$, then the LR-region is below.
2. If $angle(\Lambda) < angle(\Psi)$, then the LR-region is above
3. Otherwise the LR-region is below. [25]

3.2.4 Pruning Two Chains

Given a left polygonal chain L and a right polygonal chain R, we describe an algorithm that prunes L and R while both have more than three vertices remaining. The polygonal chains can refer to Q_L or P_L and Q_R or P_L respectively.

We have to construct the edges λ and ψ. Both are supposed to be the median edge of their respective polygonal chain. To construct a median edge, we use the index of the top and the bottom most indices of the respective chain and calculate the median index on the chain. One of the neighbours of the median index is chosen, such that this index is not the top or bottom most index. This is always possible to do, because there are at least four vertices left on the chain. The two indices then form the edge that is either λ or ψ respectively.

With λ and ψ, we compute their intersection and determine where the LR-region is located. If the edges are parallel, we determine if Λ is to the right of or on Ψ by using the intercept with the x-axis. If so, we assume that

[25] Line 308-346 in `Common.cpp`

their intersection occurs at $y = -\infty$ and that the LR-region is above the $\emptyset$-region. Otherwise, there can be no LR-region and thus no intersection and the algorithm terminates. [26]

Having determined where the intersection and the LR-region is located, it is possible to determine, if the edges border the LR-region. If the LR-region is above and the y-value of a vertex is above the intersection of ψ and λ, it must border the LR-region. Symmetrically, if the LR-region is below the $\emptyset$-region and a vertex is below the intersection, it also borders the LR-region.

The algorithm now has enough information to use the following lemmas stated by Dobkin et al. Their notation is that ψ^+ and ψ^- denote the top and bottom vertices of the edge ψ. They are also stated in such a way, that they only hold if the LR-region is above the $\emptyset$-region and only allow R to be pruned. Dobkin et al. state that symmetric lemmas can be stated easily [7]. The lemmas are shown below.

Lemma 8. *"If λ^+ and ψ^+ both border the LR-region and ψ^+ separates the vertices of λ, then the horizontal segment $\overline{\psi^+\psi^+_\Lambda}$ is a witness to the intersection of L and R." [7, Lemma 2]*

Lemma 9. *"If ψ^- lies below λ, then the problem of intersecting L and R is reduced to the problem of intersecting L and the part of R above λ^-." [7, Lemma 3]*

Lemma 10. *"If ψ^+ lies above λ, and λ^+ borders the LR-region, then the problem of intersecting l and R is reduced to the problem of intersecting l and the part of R lying below ψ^+." [7, Lemma 4]*

Lemma 11. *"If λ^+ borders the LR-region, and λ^- and ψ all border the $\emptyset$-region, then the problem of intersecting L and R is reduced to the problem of intersecting L and the part R lying above ψ^-." [7, Lemma 5]*

Lemma 8 is used to detect intersection between L and R, while Lemmas 9 to 11 are used to prune R above or below ψ. [27]

If the LR-region is below the $\emptyset$-region instead of above, symmetrical lemmas can be stated by replacing all occurrences of ψ^+ and λ^+ with ψ^- and λ^- and vice versa, and replacing all occurrences of 'above' with 'below' and vice versa. The symmetrical version of Lemma 8 that assumes that the LR-region is below becomes:

Lemma 12. *If λ^- and ψ^- both border the LR-region and ψ^- separates the vertices of λ, then the horizontal segment $\overline{\psi^-\psi^-_\Lambda}$ is a witness to the intersection of L and R.*

To state the lemmas such that they can be used to prune L, all occurrences of ψ are replaced with λ and vice versa. Additionally, all occurrences of L are replaced with R and vice versa. The symmetrical version of Lemma 8 stated for L instead of R becomes:

[26] Line 205-231 in `IntersectionDobkin2.cpp`

[27] Line 395-418 in `IntersectionDobkin2.cpp`

Lemma 13. *If ψ^- and λ^- both border the LR-region and λ^- separates the vertices of ψ, then the horizontal segment $\overline{\lambda^-\lambda^-_\Psi}$ is a witness to the intersection of R and L.*

Due to the symmetry of the lemmas, there are a total of 16 lemmas to consider for this part of the algorithm. The remaining 10 lemmas can be derived easily. Replacing symbols in the lemmas this way is essentially the same as mirroring the situation described in the lemmas along the x-axis, y-axis or both.

The lemmas are used in the following way. When Lemma 8 or one of its symmetric counterparts apply, the algorithm returns that L and R intersect. Otherwise, the other three lemmas and their symmetric counterparts are used. Note that in each iteration, only the conditions of eight of the total 16 lemmas have to be checked, because the LR-region being above or below contradicts the assumptions on which the other half depend.

Using Lemma 9 to 11, we determine what parts of L and R can be pruned. If the part of R that is above ψ^+ is not needed, the top most index of R is set to be index of ψ^+. If the part below ψ^- is not needed, the bottom most index is set to ψ^-. The same method is used for L. The algorithm then calculates if L and R still have more than three vertices each and starts a new iteration, using the reduced L and R.

Dobkin et al. state in Lemma 14, that using the lemmas in this way is sufficient to prune at least one of L or R to constant size or find an intersection, which guarantees that the algorithm will terminate at some point, and its result is correct.

Lemma 14. *"If L is a polygonal chain opening infinitely to the left and R is a polygonal chain open infinitely to the right, both having at most n vertices. The $O(\log n)$ iterations suffice either to find a horizontal segment which witnesses their intersection or to reduce the problem to detecting the intersection (or finding minimum horizontal separation) between the chains one of which has at most two edges." [7, Lemma 6]*

3.2.5 Pruning the Remaining Chain

Given two polygonal chains L and R, one of which has at most three vertices and two edges, we describe how to determine if L and R intersect. For this section, we assume that L is of constant size and R has more than three vertices. The method for the opposite case can be stated easily.

First, the state of R is stored, i.e. its top most index and bottom most index are stored. Then, for each edge λ that remains on L, we prune R (starting at its stored state) in the same fashion as described in Section 3.2.4, with one deviation: We do not prune L, because we only consider a single edge λ at a time, as if the rest of L was already pruned. Additionally, it is possible that none of Lemma 9 to Lemma 11 can be used to prune R, i.e. no progress can be made. Dobkin et al. provide additional lemmas that make it possible to progress [7, Section 2.3]. [28] The lemmas are the following two:

[28] Line 537-599 in `IntersectionDobkin2.cpp`

Lemma 15. *"If ψ^+ lies above λ, then the problem of intersection λ and R reduces to that of intersecting λ and the part of R lying below ψ^+." [7, Lemma 7]*

Lemma 16. *"If the vertices of ψ separate the vertices of λ, then the problem of intersection λ and R reduces to that of intersecting λ and the part of R above ψ^-." [7, Lemma 8]*

Like in Section 3.2.4, Lemma 16 is stated with the assumption that the LR-region is above the $\emptyset$-region. Additionally, they are both stated such that L is the chain of constant size and R is pruned. The six symmetric lemmas can easily be stated by the same method used for the lemmas in Section 3.2.5.

The proof for Lemma 15 stated by Dobkin et al. does not rely on the position of the LR-region, which means that both it and its symmetric version can be used without regard to the LR-region, i.e. if ψ^+ is above λ we may prune R above ψ^+ and if ψ^- is below λ we may prune R below ψ^-. [29]

Dobkin et al. argue that by using Lemma 8 to 11 and Lemma 15 and 16, it is possible to prune R until there are only a constant number of edges left or an intersection between L and R is found:

Lemma 17. *"If L is a polygonal chain of one edge opening infinitely to the left and R is a polygonal chain of n edges opening infinitely to the right. Then $O(\log n)$ iterations suffice either to find a horizontal segment which witnesses their intersection or to reduce the problem to detecting the intersection (or finding minimum horizontal separation) between l and a chain which has at most two edges." [7, Lemma 9.]*

Constant time. If no intersection between λ and R is found, then R will be pruned until there are at most two edges left on it. To determine if λ and the remaining edges of R intersect, we compute the intersection between λ and each remaining edge and determine if it occurs on the edges. If this is not the case Lemma 8 and its symmetric counterparts are considered to determine if intersection occurs. [30]

Iteration. If no intersection has been found up until this point, the indices of R are reset to their stored state, the other edge of L is considered as λ and the procedure is repeated.

If no intersection is found between a single edge of L and all of R, the chains L and R do not intersect.

3.2.6 Complete Algorithm

Given two convex polygons P and Q, we want to find if they intersect. The algorithm uses the binary search method described in Section 3.2.2 to find the top most and bottom most vertices of P and Q, which are used to construct the polygonal chains Q_L, Q_R, P_L, and P_R described in Section 3.2.1. We consider

[29] Line 482-507 in `IntersectionDobkin2.cpp`
[30] Line 603-632 in `IntersectionDobkin2.cpp`

the clockwise neighbours of the top and bottom most vertices, to determine if there are horizontal edges. If there are horizontal edges, the indices are increased or decreased, such that the horizontal edges are not included on any polygonal chain. Using the top most and bottom most points of both polygons, it is easy to determine if there is any vertical overlap between the polygons. If there is not, then the algorithm terminates because then they can not intersect.

Using Lemma 5, the algorithm has to determine if P_L and Q_R intersect and if P_R and Q_L intersect. To determine if P_L and Q_R intersect, let $L = P_L$ and $R = Q_R$ and use the method described in Section 3.2.3. If no intersection has been found, one of the chains is pruned to constant size and the method in Section 3.2.5 is used. If an intersection between P_L and Q_R is found, let $L = Q_L$ and $R = P_R$ and repeat the process. If both pairs intersect, then P and Q intersect, if one of the pairs does not intersect, then P and Q do not intersect either according to Lemma 5. [31]

3.2.7 Implementation Details

Indices. Dobkin et al. present an implementation of their algorithm [7], in which the polygonal chains are copied in to new arrays to simplify indexing, i.e. the topmost point of the chain is always at index 0, while the bottom most point is at the end of the array. This is done to reduce their proposed algorithm to its core and not introduce unnecessary clutter. Because this approach copies both polygons completely, their presented implementation takes $O(n)$ time, which is not the proposed $O(\log n)$ algorithm. Because the goal of this thesis is to provide a usable implementation, we have presented how to navigate the indices without copying the polygons and used this method in the implementation. Specifically, how to find the median index between two indices and how to traverse the arrays in clockwise and counter clockwise direction is described in the preliminaries in Chapter 2. [32]

Tangent edges and vertices. To be consistent with how our implementation of Barba's algorithm deals with tangent convex polygons, we have added some minor details. Let us take a detailed look at Lemma 8: The two polygonal chains intersect if both λ^+ and ψ^+ border the LR-region and ψ^+ separates the vertices of λ. Dobkin et al. state that in order for a vertex to border the LR-region, it must be strictly above or below the intersection point of Λ and Ψ. They also state that a point separates two other points only if it is strictly between them [7, Section 2.2], which is also evident in their code, see Code 3.2. Using this, polygons that only share a vertex would never intersect according to Lemma 8, because the shared vertex would never border the LR-region or separate any of the four edges it is connected to. Even worse, a polygon that is mirrored in the y-axis would not intersect with itself, because there would not exist a vertex that separates any edge. Additionally, if a vertex of λ is tangent to ψ, it is in the intersection between ψ and λ and not above or below the intersection. Therefore, we have chosen to not use strict inequality when

[31] Line 47-114 in `IntersectionDobkin2.cpp`
[32] Line 120-135 in `Common.cpp`

Code 3.1: This is a straightforward implementation of Lemma 9 to Lemma 10. This code is used to find what parts of the polygonal chain R can be pruned, by setting single bits of the `result` variable. The two inputs `lambda` and `psi` are exactly the same as λ and ψ used throughout this thesis. The inputs `intersection` and `isLRup` contain information about where the intersection between λ and ψ occurs and whether the LR-region is above or below the $\emptyset$-region. Note that the numberings of the lemmas in the comments refer to the original numbering in Dobkin [7].

```
395 int lightmind::pruneR(const Line2f& lambda,
396                      const Line2f& psi,
397                      const Intersection& intersection,
398                      bool isLRup) {
399
400     int result = DONTPRUNE;
401
402     // as written in the paper, Dobkin Lemma 3-5
403     if (isLRup) {
404         // Lemma 3
405         if (psi.bottom().y < lambda.bottom().y) {
406             result = PRUNEBELOW;
407         }
408         // Lemma 4
409         if (psi.top().y > lambda.top().y
410                 && lambda.top().y >= intersection.y) {
411             result |= PRUNEABOVE;
412         }
413         // Lemma 5
414         if (lambda.top().y >= intersection.y
415                 && lambda.bottom().y < intersection.y
416                 && psi.top().y < intersection.y) {
417             result |= PRUNEBELOW;
418         }
419     } else {
420         ...
```

implementing Lemma 8 and make use of the intersection computation for ψ and λ that has to be computed anyway.

Implementing Lemmas. Implementing Lemma 8 to 11, 15 and Lemma 16 can be done in a straightforward manner by explicitly testing the conditions of the lemmas and applying the result like in Code 3.1. Dobkin et al. present a much more compact and possibly more efficient implementation that eliminates redundant `if` statements by considering the Lemmas and their symmetric versions all at once. A conversion of their method to our code conventions is shown in Code 3.2. We have chosen to implement the version in Code 3.1 because our goal is to present a more easy to understand implementation.

Code 3.2: This code is a conversion of a code fragment presented in Dobkin [7, Section 2.2] used to determine which of Lemma 8 to 11 apply. Note that the numberings of the lemmas in the comments refer to the original numbering in Dobkin [7]. This version reduces the number of `if` statements that are necessary to use the lemmas, but trades performance for readability.

```
int LRIsUp(const Line2f& lambda,
                      const Line2f& psi,
                      const Intersection& intersection) {

    int result = DONTPRUNE;
    // if psi+ and lambda+ border the LR-region.
    if (psi.top().y > intersection.y
        && lambda.top().y > intersection.y) {
        if (psi.top().y > lambda.top().y){ // psi+ above lambda.
            if (lambda.top().y > psi.bottom().y) {
                // lambda+ separates vertices of psi.
                return INTERSECT; // Lemma 2 applies
            } else {
                result = result | PRUNE_L_ABOVE; // Lemma 4 applies
            }
        } else if (psi.top().y > lambda.bottom().y) {
                // psi+ separates vertices of lambda.
                return INTERSECT; // Lemma 2 applies
            } else {
                // symmetric version of Lemma 4 applies.
                result = result | PRUNE_R_ABOVE;
            }
    }

    // psi- below lambda-
    if (psi.bottom().y < lambda.bottom().y) {
        result = result | PRUNE_L_BELOW; // Lemma 3 applies
        // if psi+ borders the LR-region and psi+ is above lambda+
        if (psi.top().y >= intersection.y
            && psi.top().y > lambda.top().y) {
            // symmetric version of Lemma 5 applies.
            result = result | PRUNE_R_BELOW;
        }
    } else {
        // symmetric version of Lemma 3 applies.
        result = result | PRUNE_R_BELOW;
        // if lambda+ borders the LR region and lambda+ is above
            psi+
        if (lambda.top().y > intersection.y
            && lambda.top().y > psi.top().y {
            // Lemma 5 applies.
            result = result | PRUNE_L_BELOW;
        }
    }

    return result;
}
```

3.3 Sublinear Algorithm

Both Dobkins and Barbas algorithm require that polygons and their vertices are stored in clockwise order in arrays. This is required in order to do binary search on the boundaries of polygons, which both algorithms make use of. It is not always practical to store polygons in such a manner, say when elements have to be inserted or delete often, which might take $O(n)$ time each time. As an alternative, polygons can be stored in a doubly linked list, where insertion and deletion of elements can be done in constant time, but searching for an element is harder. Using Dobkin's or Barba's algorithm for a polygon stored in a linked list would require that we traverse the whole polygon in clockwise order and arrange the elements in consecutive memory, which would take $O(n)$ time. Chazelle et al. present a randomized algorithm that runs in $O(\sqrt{n})$ time and is designed to detected intersection between two polygons that are stored in doubly linked lists [3]. The algorithm requires that the elements of the linked list to be stored consecutively in order to sample random elements from it. An implementation of the algorithm is provided in `LinkedPolygon.cpp`.

3.3.1 Storage

Storing a doubly linked list in consecutive memory is straightforward. We use an array A that stores elements $a_0, a_1, \ldots, a_n$, where n is the number of vertices on the boundary of a convex polygon P. Each element a_i has four values it stores: A vertex $v(a_i)$, the index $index(a_i)$ at which it is stored in the array, the index of the clockwise neighbour $clock(a_i)$ of the vertex, and the index of the counter clockwise neighbour $counter(a_i)$ of the vertex. The array may have more space allocated than is necessary to store n elements. [33]

Insertion. When adding a new vertex v to the polygon P, we are also given an index i of the element that v should be clockwise to. We look up a_i and $a_{clock(a_i)}$. Then we create a new element and store it at the first free position in the array, which will be at $n-1$. In it, we store the vertex, its index in the array, let a_i be it counter clockwise neighbour, and let $a_{clock(a_i)}$ be its clockwise neighbour. Then we update $clock(a_i)$ to be $n-1$ and $counter(a_{clock(a_i)})$ to be $n-1$. Thus, inserting a new element only required to change two indices (which are essentially pointers), and creating a new element, which is done in constant time. For more details on linked lists see [4, Chapter 10.2]. [34]

If the array has no free space in which the new element can be stored, a new array with twice the size is created and all elements from the old array are copied in to the new array and the old array is deleted. This technique takes amortized linear time in the number of insertions [4, Chapter 17.4, p. 463].

Random Sampling. Because all elements are now contained in an array and stored consecutively, though in no particular order, it is possible to choose random vertices on the boundary of P, by selecting a random index between

[33] See `LinkedElement.cpp`

[34] Line 29-51 in `LinkedPolygon.cpp`

0 and $n-1$. The algorithm presented by Chazelle et al. requires that we are able to sample r elements. To do this, we can use a well known variation on the Fisher-Yates shuffle [4, p.126].

We can randomly select r elements with the following method. Maintain a counter c. While $c < r$, randomly choose an index j from 0 to $n-c-1$, then exchange the elements at index j and index $n-c-1$ while maintaining all references to them. This will place randomly chosen elements at the end of the array, and will not allow selected elements to be chosen again. The last r elements in the array are then the sample. [35]

3.3.2 Algorithm

The algorithm presented by Chazelle et al. is as follows. We are given two convex polygons P and Q that are stored in doubly linked lists in consecutive storage and are asked whether or not the the two polygons intersect. Start by taking a random sample R_p of $\sqrt{size(P)}$ vertices from P and a random sample R_q of $\sqrt{size(Q)}$ vertices from Q. We now have that $\text{CH}(R_P) \subseteq P$ and $\text{CH}(R_Q) \subseteq Q$.

The algorithm now determines if $\text{CH}(R_P)$ and $\text{CH}(R_Q)$ are separable, without constructing the boundaries explicitly. This can be done in expected linear time in the input with a linear programming algorithm. We describe the details of the linear programming algorithm in Section 3.3.4. For now it is enough to know that we have a method to find a separating line and an element on $\text{CH}(R_P)$ and $\text{CH}(R_Q)$ each, which are tangent to the separating line. If no separating line exists, the linear programming algorithm reports that the input is not separable. [36]

If $\text{CH}(R_P)$ and $\text{CH}(R_Q)$ are not separable, P and Q intersect and the algorithm terminates. If the two hulls are separable, we have an element p on P and an element q on Q, whose vertices both are tangent to the separating line $\mathcal{L}$. Let p_1 and p_2 be the two neighbours of p on the boundary of P. If none of p_1 or p_2 is on the same side of $\mathcal{L}$ as $\text{CH}(R_Q)$, let C_P be an empty set of vertices. Otherwise, only one of p_1 or p_2 can be on $\text{CH}(R_Q)$'s side of $\mathcal{L}$. Then, we can walk along the boundary of P and collect all elements that are on the same side of $\mathcal{L}$ as $\text{CH}(R_Q)$. Let this chain of vertices be C_P, see Figure 3.14. A similar chain C_Q can be constructed from Q using q and its neighbours. [37]

Chazelle et al. state that $P \cap Q \neq \emptyset$ if and only if P intersects C_Q or Q intersects C_P [3]. To test if P and C_Q intersect, we do the following. We use linear programming again to find a separating line between R_P and C_Q. If there is no separating line, P and Q intersect and the algorithm terminates. Otherwise, let $\mathcal{L}'$ be line that separates R_P and C_Q. Like before, we use $\mathcal{L}'$ to construct the part of P that is on the other side of $\mathcal{L}'$ and call it C'_P. Now test C_Q and C'_P for intersection using linear programming. If the two intersect, the algorithm terminates. [38]

[35] Line 151-171 in `LinkedPolygon.cpp`
[36] Line 260-275 in `LinkedPolygon.cpp`
[37] Line 174-254 in `LinkedPolygon.cpp`
[38] Line 277-338 in `LinkedPolygon.cpp`

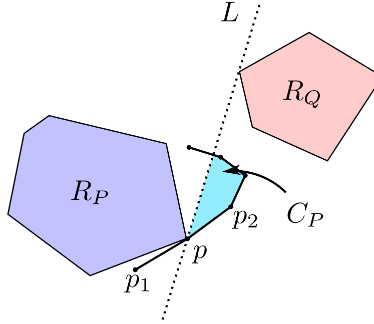


Figure 3.14: The shapes $\mathrm{CH}(R_P)$ and $\mathrm{CH}(R_Q)$ and their separating line L. The chain C_P, which contains all vertices that are on the other side of L, can be found by starting at p and by traversing the boundary of P along the direction in which a neighbour of p lies outside.

If P did not intersect C_Q, we still have to test if Q intersects C_P. This is done in the exact same manner. We use linear programming to test if R_Q and C_P intersect, and if they do not, we have found a separating line $\mathcal{L}'$, which we use to construct C'_Q. Then we use linear programming to determine if C'_Q and C_P intersect. [39]

If none of these constructions intersect, then P and Q do not intersect and the algorithm terminates.

Correctness and running time. Chazelle et al. state that this algorithm is expected to run in $O(\sqrt{n})$ time and will report the correct result, according to the following theorem.

Theorem 1. *To check wether two convex n-gons intersect can be done in $O(\sqrt{n})$ time, which is optimal. [3, Theorem 1.2]*

The reasoning for this running time is not immediately obvious, because we have to traverse the boundary of P and Q to find C_P, C'_P, C_Q, and C'_Q, which might contain any number of elements. Chazelle et al. prove that the expected size of these chains is $O(n/r)$, where r is the size of the sample which is chosen as $r = \sqrt{n}$ [3, Theorem 2.1, Formula (2)].

3.3.3 Implementation Details

The algorithm presented by Chazelle et al. is straightforward to implement (apart from linear programming), with only few parts that need additional consideration due to possible numerical instability.

Sidedness. Consider we have found a separating line $\mathcal{L}$ between R_P and R_Q, that is tangent to p and q. To construct C_P and C_Q, we need to determine on which side of the line R_P and R_Q are. To decide sidedness, we take a constant number of vertices in R_P and compute their sidedness in relation to $\mathcal{L}$, and report the sidedness of the majority. This should avoid that a single element that is very close to L can report a false sidedness. It would also be possible to

[39] Line 342-398 in `LinkedPolygon.cpp`

find the vertex that is furthest away from L and report its sidedness, though distance computations are more costly in terms of processing power.

Constructing chains. Having decided the sidedness of R_P and R_Q, the algorithm has to construct the chain C_P. To do this, we look at the neighbours of p. Let p_1 be the clockwise neighbour and p_2 be the counter clockwise neighbour. Because p_1 and p_2 may be very close to p and therefore close to L, their sidedness might be computed erroneously. To mitigate this, one can travel a constant number of elements along the boundary in each direction, and use a majority vote to determine which side of the line one is on. Then, constructing the chain is easy. We traverse and collect elements along the boundary, until we are on our own side again.

Tangent edges and vertices. Like with Dobkins and Barbas algorithms, we would like to detect intersection between P and Q, if they have a vertex in common, or a vertex is tangent to an edge of the other polygon. If the linear programming part of the algorithm does not detect such cases, e.g. it allows a separating line to go through two vertices that are equal and allows vertices to be placed on the separating line, we must detect these cases after a separating line is found between C_P and C'_Q or C_Q and C'_P. To do this, we can consider the two edges of P that p is connected to, and the two edges of Q that q is connected to. If two vertices are equal, the separating line must intersect them both, and either one p and q are intersecting or a vertex on an edge they are part of is intersecting, thus testing those four edges for intersection will reveal this case. The same work, if a vertex intersects with an edge, in which case $\mathcal{L}$ is tangent to the intersecting edge and either p or q is the intersecting vertex.

3.3.4 Linear Programming

The algorithm presented by Chazelle et al. achieves its $O(\sqrt{n})$ running time by using a randomized linear programming algorithm to compute a separating line between two sets of points of size $O(\sqrt{n})$.

In this section, we will give a short summary of the linear programming problem and techniques that are used to solve them. We then present details necessary to implement a linear programming algorithm that searches for a separating line between two sets of points in Section 3.3.4. An implementation that searches for a separating line between two sets of points can be found in `linearProgramming.cpp`.

Overview of linear programming.

Linear programming algorithms are capable to find a solution to a set of linear constraints, while maximizing a linear cost function [5, Chapter 4.3, p.71]. Such a linear optimization problem is described as follows:

$$\text{Maximize} \quad c_1x_1 + c_2x_2 + \cdots + c_dx_d$$

$$\text{Subject to} \quad \begin{aligned} a_{1,1}x_1 + \cdots + a_{1,d}x_d &\leq b_1 \\ a_{2,1}x_1 + \cdots + a_{2,d}x_d &\leq b_2 \\ &\vdots \\ a_{n,1}x_1 + \cdots + a_{n,d}x_d &\leq b_n \end{aligned}$$

where the c_i, and $a_{i,j}$, and b_i are real numbers, which are the input to the algorithm [5, p.71]. The parameter d is the dimension of the linear program, and the function to be maximized is called the objective function. The goal is to find a point $(x_0, \ldots, x_d)$ that maximizes the objective function and satisfies all constraints. Intuitively, each constraint can be understood as a half-space in $\mathbb{R}^d$, and the intersection of all those half-spaces contains a solution to the linear program. If the intersection of all those half-spaces is empty, there is no solution and we say that the problem is infeasible.

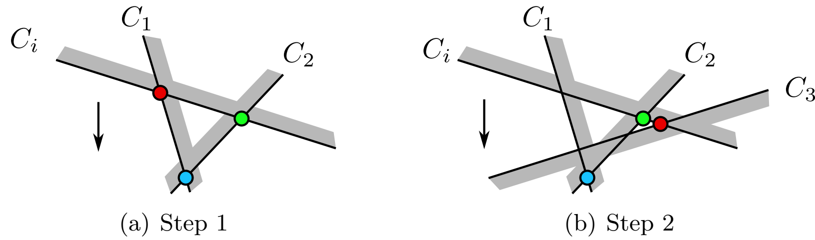
Simplification. Due to the nature of the linear optimization problem we are going to solve later, we can restrict ourselves to the problem of bounded linear optimization problem, i.e the objective function can not be infinitely large. A bounded linear optimization problem is given two constraints as an input, that bound the solution such that the objective function can not be infinite. Let the two constraints be m_1 and m_2. We can also limit ourselves to the case there $d \leq 2$, because we are only going to solve problems in the plane, which means that all constraints will effectively be lines, that split the plane into a feasible and a non feasible part.

Intuition of the linear programming algorithm. The linear optimization problem can the be solved as follows: Use m_1 and m_2 to find an initial solution, we call this our current guess. Randomly permute the list of constraints. Go through each constraint one by one. If the current guess satisfies the constraint, do nothing and go to the next constraint. Otherwise, the constraint is violated and we have to update our guess. To do this, we consider each intersection point between the current constraint and every constraint we have looked at until now, and keep track of the two points that yield the largest feasible and the smallest feasible value of the objective function. If the largest value would be smaller than the smallest value, then there exists no point that satisfies the constraints and the problem is infeasible, otherwise we let our guess be the point that yields the largest value of the objective function. If all constraints are processed, report the current guess. For an illustration of this process see Figure 3.15.

This section covers the general method of of linear programming. Details specific to our implementation are presented in the next section.

Finding a Separating Line

To use a linear programming algorithm to find a separating line, we must state the problem of finding a separating line as a linear optimization problem. Conceptually, we want to find a line $y(x) = ax + b$ such that the points of one



(a) Step 1 (b) Step 2

Figure 3.15: When the linear programming algorithm finds a constraint C_i, that is violated by the current guess, it examines every constraint that has been satisfied so far. The black arrow denotes the direction in which the objective function would be maximized. The grey regions along the line denote the half-space that contains feasible points for that constraint. In 3.15(a), we have the current guess as the blue point, and have already examined the two constraints C_1 and C_2 and found the intersection between them and C_i. The red intersection point is the point that has the smallest value of the objective function on C_i, and the green intersection point has the largest value of the objective function. The next step is to examine constraint C_3 in 3.15(b). The constraint C_3 restricts points on C_i that lie in opposite direction to the objective function, and therefore must the intersection between C_3 and C_i have the smallest possible value. Now the smallest possible value is larger than the largest possible value, which was bounded by C_2, and there exists no feasible region anymore.

polygon are above it, and the points of the other polygon are below it. This means that a and b are the two variables that form the 'point' we want to find with the linear program. We call a the slope of the line and b the intercept of the line. We also want the line to be tangent to two points, one of each polygon. Additionally, we need two constraints that bound a and b such the objective function can not approach infinity. At last, we have to keep track of the two points that are tangent to the separating line and deal with infeasible cases.

Constraints. First, we describe how to express that one set of points should be above and below the line. Let A be the set of points that should be above the line and let x_i and y_i be the x and y values of the i'th point in the set. If $y_i \geq y(x_i) = ax_i + b$, then i'th point is above the line and we get the constraint $C_i : ax_i + b \leq y_i$. Let B be the set of points that should be below the line and let x_j and y_j be the x and y values of the j'th point in the set. If $y_j \leq y(x_j) = ax_j + b$ then the point is below the line. The constraint is then $C_j : -ax_j - b \leq -y_j$. Note that all x_i, y_i, x_j, and y_j are constants, because they are defined by the input to the algorithm, while a and b are what we are trying to find.

Conversions. Note that every constraint C_j and C_i defines a line, in that we have two variables a and b that are dependant, while the point from A or B is fixed. Thus we could compute a as a linear function of b and vice versa for any given constraint, e.g. using $C_i : b = -x_i a + y_i$. This way, we can think of all points in A and B as lines.

Intersections between constraints. Considering constraints as lines or functions of a and b, it is possible to compute the intersection between two

constraints using two equations with two unknown variables, which will result in a point in terms of a and b, which again represents a line $y = ax + b$.

Objective function. Only using these constraints, we would find some separating line, but we need a separating line that is tangent to two points. This is what the objective function is used for. We define the objective function to be $-a$, i.e. we want to minimize the slope of the line. This will force the line to always being tangent to at least two points. If the line would only be tangent to one point, we could trivially find a line with a smaller slope by rotating it around the point it is tangent to, and the linear optimization problem would not be solved.

Bounding the solution. Choosing $-a$ as the objective function may result in that the linear program is unbounded, because if the leftmost point in A is to the right of the rightmost point in B, there exists a vertical line that separates A and B. To prevent this, we scan A and B before we use linear programming, to find the left most and right most points in both sets. If A and B are separated by a vertical line, it is enough to relabel A and B, such that $B^* = A$ and $A^* = B$ is to the left of the vertical line. Otherwise let $A^* = A$ and $B^* = B$.

Consider the following: If the slope is negative, i.e. $a < 0$, then the elements of B must be to the left of the line and A to the right, but if $a > 0$, then the elements of B must be to the right and the elements of A to the left of the line. If a can be arbitrarily negative, then B is to the left of the vertical line, and A to the right of the vertical line. If we assume that a could be arbitrarily positive, we have a contradiction, because then B would be to the right of the vertical line separating A and B. This means that a can either be arbitrarily negative or positive, but never both (unless one of A or B is empty).

By relabeling A and B such that A^* is always to the left of a vertical separating line, slope a could be arbitrarily positive, but not arbitrarily negative, which means that we can find a minimum value for a.

Initial constraints. The algorithm requires two constraints that will always bound a. Because we have chosen to relabel A and B if A's left most point is to the right of B's rightmost points, we know that A^*'s leftmost point is to the left of B^*'s rightmost point. Thus we can use those two points for the initial guess of a separating line. Note that we can not use two arbitrary points of A and B as the initial constraints for the solution, because we might choose a point of A^* that is to the right of the point from B^*, in which case a would approach negative infinity again.

Points tangent to the separating line. Every time a constraint is violated, the separating line is updated, if it still exists. Every time we consider an intersection between two constraints, we can check if the constraints are from both A and B, and not from the same set. If they are from different sets and

limit how large the objective function can be, we store the two constraints. If the separating line is reported, we can also report the two stored points.

Infeasible cases. When computing a separating line, the algorithm may find that it is impossible to choose a line that satisfies all constraints, in which case the problem is infeasible. There are two cases where this can happen. Either no line exists that separates A and B, i.e. the shapes intersect, or there exists a separating line, but A is below it and B is above it. To prevent a false negative result, one can run the linear programming algorithm twice, swapping the labels of A and B. Because we search for left most and right most points anyway, one can search for the bottom most and top most points at the same time and construct the bounding boxes of both sets and detect such a case in advance or even if they trivially intersect.

3.3.5 Implementation Details for Linear Programming

Input. Our implementation of linear programming is given two lists of elements, that are originally elements of a doubly linked list as described in Section 3.3.1. This means we do not only have access to the vertices on the boundary of polygons, but also their position in the input to the algorithm described in Section 3.3.2. Every time we update the separating line, we store these indices of the two vertices and return them when the linear programming algorithm finishes. This makes it easy to construct the chains needed for the algorithm in Section 3.3.2, as we do not have to search for the vertices on the boundary.

Randomization. The randomized linear programming algorithm we use, should randomize its input to get the expected linear running time. Instead, we let the method that calls our linear programming algorithm randomize the input. We have chosen to do this, because the algorithm in Section 3.3.2 already provides randomized input in the form of R_P and R_Q, which are random samples of P and Q. Randomizing those again would only waste computation time. Only the chains C_P, C'_P, C_Q, and C'_Q are randomized before they are used in the linear programming algorithm.

The linear program now has two randomly permuted lists as its input and both lists represent a list of constraints. This deviates from the original linear programming algorithm in [5, Chapter 4], where only a single randomly permuted list of constraints is present. We have not merged both lists of element in to one, because we would lose information about which set of points the element belongs to. To retain this information, one could wrap elements in to objects or have each element store which polygon it belongs to from the beginning. Instead, we have chosen to switch between the two lists, when examining new constraints, so that when a constraint from set A is examined, a constraint from set B will be examined in the next iteration.

Chapter 4

Experiments

In this chapter, we will compare the performance of the three implemented algorithms. We have asymptotic bounds of $O(\log n)$ and $O(\sqrt{n})$ for the algorithms, but have no analysis of the constant factors that hide in the asymptotic notation. By measuring and comparing the running times of the algorithms, we can verify that they run according to their asymptotic bounds and may be able to recommend the most efficient algorithm.

Because the algorithm in Chapter 3.2 and Chapter 3.1 work on fundamentally different input than the algorithm from Chapter 3.3, we will compare them to each other first, before we examine the performance of the last algorithm.

4.1 Setup

We describe the machine on which experiments are run, and how we measure time during the experiments in this section. We also describe how we generate input for the first experiment.

The machine The computer on which all experiments are run is a Lenovo L430. It has an Intel i5-3230M CPU @ 2.60GHz with two physical cores and access to 8 GiB of ram.

Code. All code is written in `C++11` and compiled using `gcc 4.8.4` under Ubuntu 14.04. The `gcc -O3` flag is used, to let the compiler optimize the program.

Measuring Time. To measure the time it takes for an algorithm to compute an intersection between two convex polygons, we use `std::chrono::high_resolution_clock` from the `C++11` standard library. This clock is precise enough to measure times in the order of dozens of nanoseconds. A measurement is taken by starting the clock, computing the intersection between the same two polygons one thousand times, stopping the clock, and dividing the resulting time by one thousand to get an average running time. Each measurement is repeated one hundred times with new randomly generated polygons. Each set of polygons that is generated is used with all algorithms, and the results are

compared, to make sure that the algorithms agree on the result. If the algorithms disagree about a result, the measured running times are discarded and we are notified that we have to debug the implementations.

Early termination. The running time of the algorithms may depend on if the polygons intersect. Therefore we measure the running time for intersecting and non-intersecting polygons separately. For example Barba's algorithm has several cases where early termination is possible, while Dobkin's algorithm can terminate earlier, if the polygons do not intersect, because the first two polygonal chains that are examined may not intersect and the algorithm never has to look at the other pair of polygonal chains.

4.1.1 Starting Point for Generating Inputs

To generate a convex polygon, we randomly choose two parameters $a \in [0, 1]$, and $b \in [0, 1]$, that define an ellipse $(\frac{x}{a})^2 + (\frac{y}{b})^2 = 1$. We then randomly choose n points on the ellipse by using $x = a\cos\theta$ and $y = b\sin\theta$, where $\theta \in [0, 2\pi]$. This is the equivalent of choosing uniformly random points on the unit circle, and then stretching or squishing the circle, see Figure 4.1 for an example. Then, all n vertices are translated by $\Delta x \in [-1, 1]$ along the x-axis and by $\Delta y \in [-1, 1]$ along the y-axis. The ellipses are translated, because they would trivially intersect, as the origin is part of all ellipses chosen in this way. The incremental convex hull algorithm [5, p.6], also known as Graham-Scan, is used to order the vertices along the hull and remove any degenerate vertices and to ensure that the vertices of the polygon are stored in clockwise order.

Using ellipses to generate convex polygons is supported by the fact that any convex shape in the plane can be approximated by an ellipse. This is due to the following theorem.

Theorem 2 (John's Lemma). *[8] Let $K \subset R^n$ be a bounded closed convex body with nonempty interior. Then there exists an ellipsoid E_{in} such that*

$$E_{in} \subseteq K \subseteq E_{out},$$

where E_{out} is E_{in} expanded from its center by the factor n. If K is symmetric about the origin, then we have the improved approximation

$$E_{in} \subseteq K \subseteq E_{out} = \sqrt{n} \cdot E_{in}.$$

This theorem tells us, that if we were to generate convex polygons in some other way, the result would be something that is similar to ellipses, in which case we might as well have used ellipses to start with.

4.2 Comparing the algorithms of Dobkin and Barba

In this section, we measure and compare the running times of Dobkin's algorithm and Barba's algorithm. Both algorithms have a asymptotic bound of $O(\log n)$ and we would like to verify that this holds in practice.

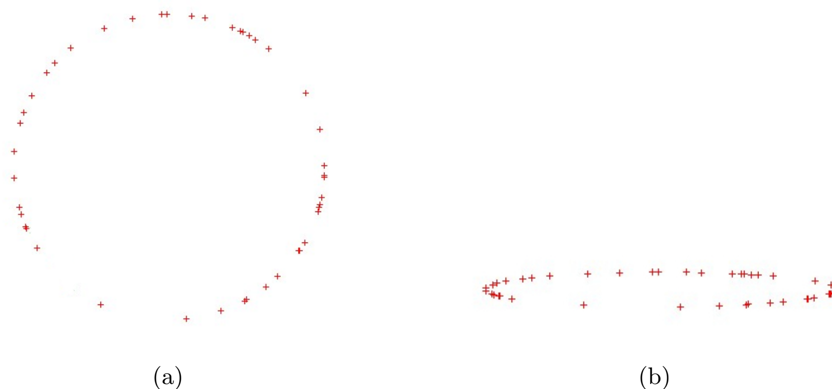


Figure 4.1: Figure 4.1(a) shows randomly generated vertices on the unit circle. The vertices are then squished by using $a = 1$ and $b = 0.1$ in Figure 4.1(b). This results in that the sharp turns of the ellipse, i.e. details, are densely populated with vertices, while the rest of the ellipse is more sparsely populated.

Expectation. The algorithms are expected to run in $O(\log n)$ time. Additionally, the implementation of Barba's algorithm requires more intersection computations and sidedness tests per iteration that Dobkin's algorithm, which might be more computationally expensive than the comparisons of y-coordinates that are used for the lemmas of Barba's algorithm.

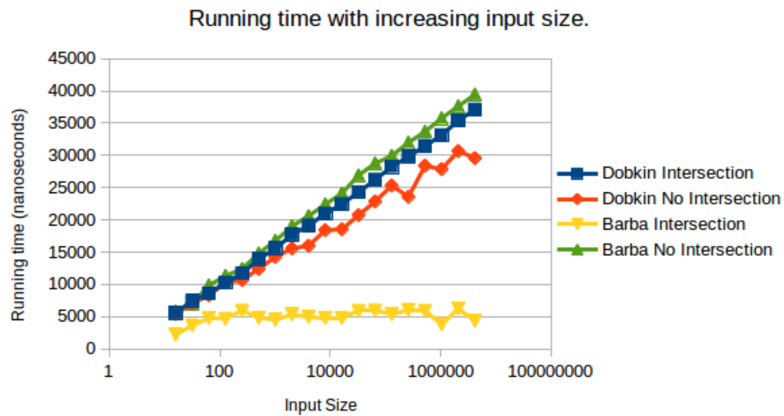
4.2.1 First Experiment

To verify our expectation, we test the algorithms with input sizes from $n = 2^4$ to $n = 2^{22}$. Both input polygons are of the same size $size(P) = size(Q) = \frac{n}{2}$. The input size doubles each time measurements for a certain input size have been made. The resulting measurements are shown in Figure 4.2.

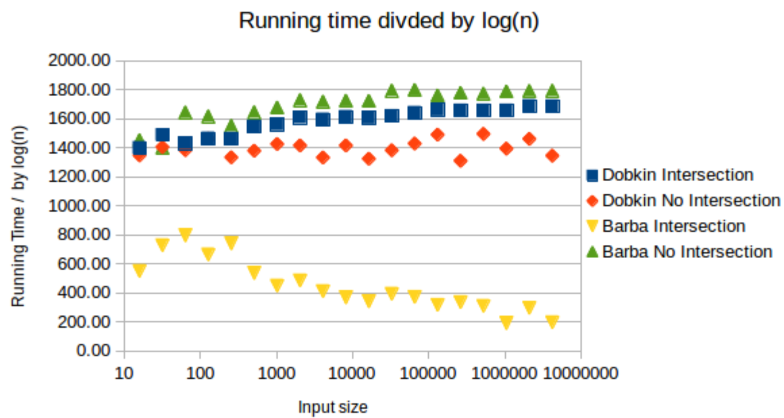
Discussion. Observing the running times in Figure 4.2(a), it is easy to see that Barba's algorithm and Dobkin's algorithm perform very similarly, with one big exception: If the polygons are intersecting, Barba's algorithm performs much faster than all other cases and the running time barely increases along with the size of the input, such that the algorithm seemingly runs in constant time.

To verify that the running times conform to the $O(\log n)$ bound, they are divided by $\log(size(P) + size(Q))$ in Figure 4.2(b), in which we see that both algorithms conform to their worst case bound.

Contrary to our expectation, Barba's algorithms is on average faster that Dobkin's algorithm. It is possible that this is caused by the number of iterations the algorithm have to perform. Barba's algorithm can never use more than $\log(size(P)) + \log(size(Q))$ iterations, while Dobkin's algorithm can use up to $2\log(size(P)) + 2\log(size(Q))$, because both pairs of polygon chains may have to be traversed.



(a) Raw running time



(b) Running time divided by $\log n$

Figure 4.2: Measured running times for Barba's algorithm and Dobkin's algorithm, for increasing sizes of input. The running time for instances where the two input polygons intersect are measured separately from inputs that do not intersect. Figure 4.2(a) shows the averaged raw running times in nanoseconds for both algorithms. The running times are divided by $\log(size(P) + size(Q))$ in Figure 4.2(b).

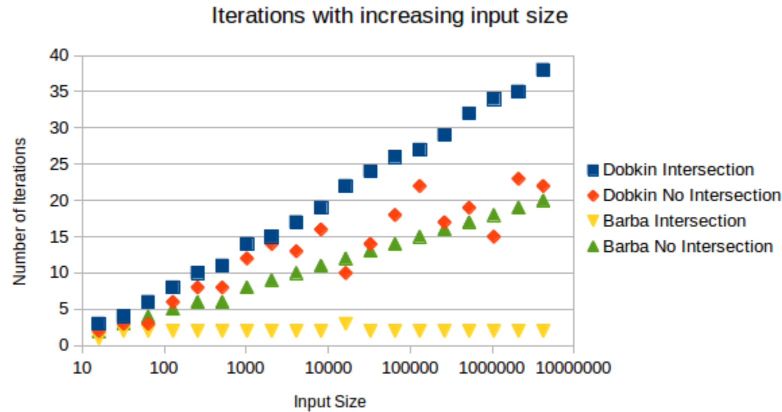


Figure 4.3: The number of iterations each algorithms uses. Dobkin's algorithm uses generally more iterations, because it has to work on half the input twice, once for each pair of polygonal chains, whereas Barba's algorithm works on the complete input.

4.2.2 Second Experiment: Iterations

To find out why Barba's algorithm is so much faster when the polygons intersect count the number of iterations both algorithms use to return a result. This may also give some insight into why Barba's algorithm performs as fast as Dobkin's algorithm, even when the polygons do not intersect. To count the number of iterations, we increase a counter every time a loop that prunes part of the input is entered.

Discussion. Counting the number of iterations in Figure 4.3 reveals that Barba's algorithm on average does not iterate more than four or five times if the polygons intersect. To make sure that this is not due to the `gcc` compiler optimizing the code, we repeated the experiment without any optimization flags, but the number of iterations did not change significantly. Another factor that might have influence on these results is that all polygons are oriented in the same way, i.e. the polygons are never rotated and the first vertex in a polygons array is always the leftmost vertex. Apart from that, we observe that Dobkin's algorithm usually uses more iterations than Barba's algorithm does, especially when the polygons intersect. in the worst case. This stems from the fact that Dobkin's algorithm has to search through both pairs of polygonal chains if the polygons intersect, which obviously doubles the number of iterations it takes to get a result. Looking at the comparable running times of both algorithms in Figure 4.2 and the obvious difference in the number of iterations, we can surmise that a single iterations of Barba's algorithm is about twice as expensive as a single iteration of Dobkin's algorithm in term of running time. We also observe that Barba's algorithm seemingly never uses early termination when the polygons do not intersect, as the number of iterations closely follows a $\log n$ curve very well.

4.2.3 Third Experiment

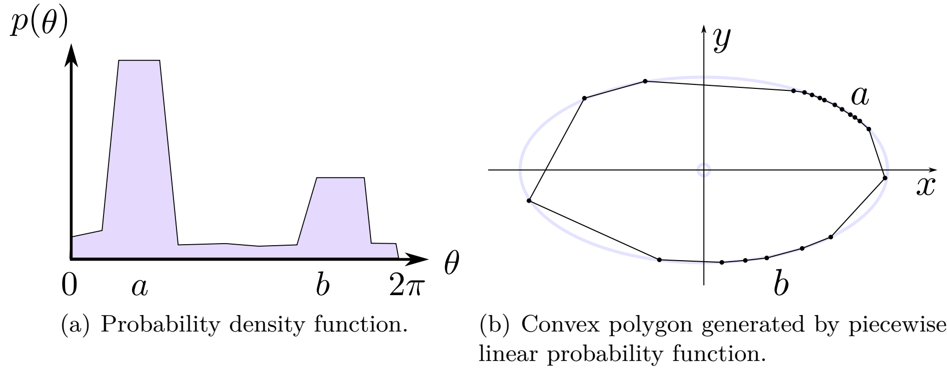
Because the input we used for the first two experiments might be to easy or simple, we decided to change how the input is generated. We change how polygons are generated, by rotating them around the origin before they are translated and before the convex hull algorithm is used. The indices are then rotated, such that the first vertex in the array is some random vertex on the polygon instead of the leftmost vertex. Using this way to generate random convex polygons still does not change the outcome of the experiments in any significant way. In the next section, we further change how polygons are generated.

4.2.4 Fourth Experiment: Non-uniformly Distributed Vertices

Up until now, all vertices were distributed uniformly along the boundary of polygons. Because of this, the input may be too easy for Barba's algorithm, leading to the very fast running times we observe. To generate polygons whose vertices are not uniformly distributed, we use a random number generator from the `C++11` standard library which generates numbers according to a piecewise linear probability distribution called `std::piecewise_linear_distribution` [1]. Such a distribution is constructed by assigning weights to values in the domain in that we want to generate numbers, i.e. we can assign two weights w such that $w(1) = 5$ and $w(2) = 10$, then of choosing 2 is twice as probably as choosing 1, while the probability of elements in between 1 ans 2 is a linear function between the two. An example of such a distribution in use is shown in Figure 4.4. To further randomize how polygons are generated, we do not use one single probability distribution function, but randomly generate distribution functions for polygons, by splitting the interval $[0, 1]$ into a random number of segments of uniform length and assigning each segment a uniformly random weight.

Discussion. Using polygons that are generated by using non uniform distribution functions, rotation and shifted indices does not change the number of iterations or running times of the algorithms in any significant way. To make sure that different kinds of input are generated, we have visualised several inputs during the course of the experiments, two of which are included in Figure 4.5.

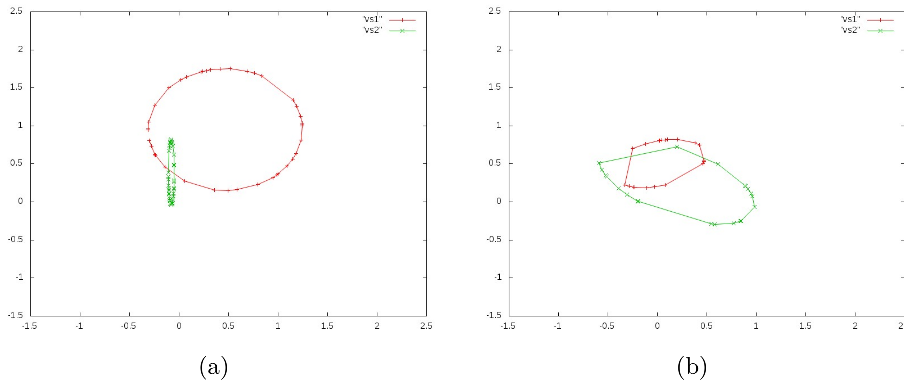
[1] For more details, see the API at `http://en.cppreference.com/w/cpp/numeric/random/piecewise_linear_distribution`



(a) Probability density function.

(b) Convex polygon generated by piecewise linear probability function.

Figure 4.4: This figure illustrates what the result of using a piecewise linear probability function to generate a polygon might look like. Figure 4.4(a) illustrates the probability function over angles from 0 to 2π, where the angles around a and b have high probabilities of being chosen, while all other angles have very low probabilities of being chosen. When using this probability density function to choose several angles to generate points on an ellipse, we might generate a polygon like that shown in Figure 4.4(b), where most of the boundary is sparsely populated by vertices, but the angles close to a and b are densely populated.



(a)

(b)

Figure 4.5: The polygons in Figure 4.5(a) and Figure 4.5(b) are generated by using two different constructions. All polygons have the same number of vertices. The first uses uniformly generated angles on the unit circle to generate vertices on an ellipse and then translates all vertices. The second one uses non uniform probability density functions to generate angles on the unit circle, rotates and translates the polygon, and rotates the indices in the array, such that the leftmost vertex is not the first vertex in the array. These differences can be easily spotted; in Figure 4.5(a), the vertices on the red polygon are distributed fairly evenly while vertices on the green polygon are squished together at the sharp turns, while the vertices in Figure 4.5(b) are much more clustered and have more sparse areas on the polygons boundaries. Also notice that the red polygon in Figure 4.5(a) has a small hole in the boundary on its left side. This is where the first and last vertex of the array are located, which is not the case for the polygons in Figure 4.5(b), whose first and last indices can be anywhere on the boundary.

4.3 Running time for the Sublinear Algorithm

In this section we measure the running times of the sublinear algorithm presented in Chapter 3.3 and verify that it runs in $O(\sqrt{n})$ time. Then we compare the sublinear algorithm to Dobkin's and Barba's algorithm. Additionally, we compare its performance to the naive algorithm, that uses linear time to convert a doubly linked list representation to an array representation and uses a binary search algorithm to detect intersection.

Generating input. To generate input in the form of a doubly linked list as described in Chapter 3.3.1, we use the method described in Section 4.2.4 and then modify it in the following way. Given a polygon whose vertices are stored in clockwise order in an array, we can trivially construct a doubly linked list that stores the vertices in the same order, by inserting the elements one by one between the last element and the first element in the doubly linked list. We then shuffle the elements in the same way the random sampling method described in Section 3.3.1 does, with the only difference being that we shuffle all elements in the linked list.

4.3.1 Fifth Experiment: Verify the Asymptotic Running Time

To verify the expected $O(\sqrt{n})$ running time of the algorithm, we run it on two randomly generated polygons of size 2^4 to 2^{20}. We reduced the number of repetitions done between starting and stopping the measurement of time to 10 for input sizes above 2^{18}, but do still use 100 pairs of polygons for each input size. This has been done because the sublinear algorithm runs much slower on large input. The resulting running times divided by the square root of the input size are shown in Figure 4.6.

Discussion. We can observe that the running times divided by $\sqrt{n}$ nicely converge towards two constants. This strongly indicates that the algorithm performs according to it expected asymptotic running time of $O(\sqrt{n})$. Additionally, we see that there is a large difference in performance depending on if the polygons intersect or not. If they do intersect, the algorithm returns approximately 4 to 5 times quicker then if they do not intersect. This can easily be explained by the use of the linear programming algorithm. If the polygons intersect, it seems very likely that the algorithm will not find a separating line between R_P and R_Q and thus terminate immediately after running the linear programming once. If the polygons do not intersect, the linear programming algorithm is run five times on input that is expected to have size of equal order of magnitude. This seems to explain the difference in running time very well.

4.4 Comparing the Algorithms.

We have confirmed that the implementations of all three algorithms perform according to their expected asymptotic running time. From this, we can expect

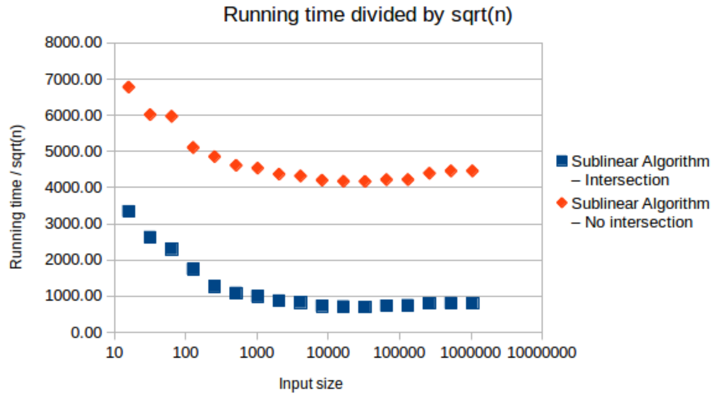


Figure 4.6: Running time of the sublinear algorithm, i.e. the algorithm presented by Chazelle et al., divided by the square root of the input size. The running time was measured in nanoseconds. The blue dots are measurements where the two input polygons did intersect, while the red dots represent measurements where the polygons did not intersect. We can observe that the running time depends a lot on the output of the algorithm and that dividing the running time with $\sqrt{n}$ converges to a constant, which supports that the algorithm runs in expected $O(\sqrt{n})$ time.

that the sublinear algorithm performs slower that Dobkin's or Barba's algorithm. We confirm this by plotting the running times of all algorithm in Figure 4.7.

Seeing that the sublinear algorithm performs much worse, we can consider if it would be faster to convert polygons stored in linked lists to polygons stored in sorted order and run Barba's algorithm instead. This would be a linear time algorithm because the whole input is scanned once, but scanning the whole input might be faster than running linear programming five times, at least for small inputs. We show a comparison between this naive conversion and the sublinear algorithm in Figure 4.8, and find that it is indeed faster to convert linked lists, if the inputs have less than 2^{10} elements. Beyond that points, converting the linked lists to ordered arrays is extremely slow compared to just running the sublinear algorithm, because of the linear asymptotic time of the conversion.

4.5 Concluding the Experiments

Barba's algorithm performs very well, especially when the input polygons do intersect, in which case it performs in nearly constant time on average. Dobkin's algorithm performs well in comparison, using slightly less time than Barba's algorithm when the polygons do not intersect. In general, these two algorithm terminate after less than 50 microseconds, even with inputs that consists of millions of vertices, which is very efficient with Barba's algorithm being the most efficient on average.

If storing polygons in clockwise order is not feasible and linked lists are required, the algorithm presented by Chazelle et al. is a reasonably efficient solution even for large inputs. The measured running times for inputs with

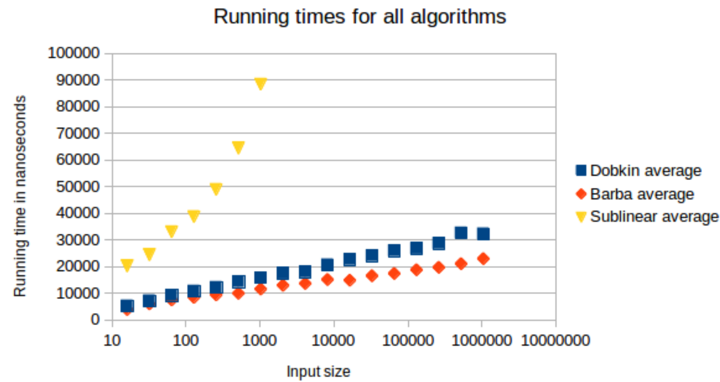


Figure 4.7: We plot the average running times of all three algorithm to compare them. It is obvious that the sublinear algorithm is much slower than the binary search algorithms.

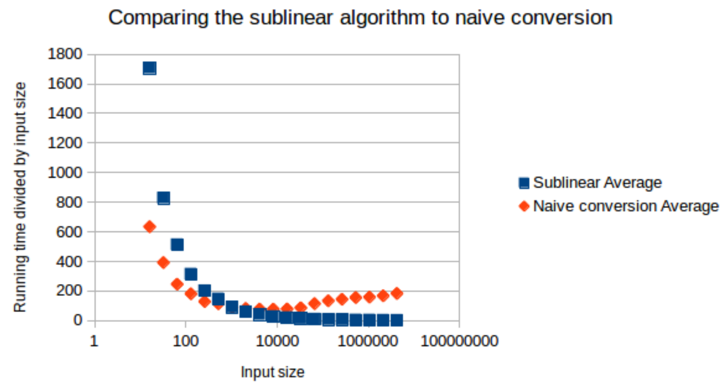


Figure 4.8: A comparison of the performance of the sublinear algorithm with a naive solution, that traverses linked lists and turns them in to ordered arrays and then uses Barba's algorithm to determine intersection. Using the naive solution is faster for inputs that are small, but otherwise is the algorithm by Chazelle et al. $O(\sqrt{n})$ more efficient.

millions of vertices is in the order a few milliseconds, which is much better than the alternative naive algorithm that turns a linked list to an array, which is orders of magnitude slower.

Chapter 5

Conclusion

We have discussed three algorithm that determine if two convex polygons intersect. The first two algorithms require that polygons are stored in an array with vertices in sorted order to be able to use binary search techniques. The third algorithm only requires the polygons to be stored in linked lists in successive memory and uses randomization techniques. All three algorithms have working implementations that accompany this thesis [1].

The first algorithm is originally presented by Barba et al. [1]. This algorithm is a binary search algorithm, yielding an asymptotic running time of $O(\log n)$. We present an overview of their algorithm and provided many additional details, which can be used as guidelines to implement an efficient version of their algorithm. We show how a separating line between two convex shapes with a constant number of edges can be found easily. We also expand upon the proofs by Dobkin et al. to show that the algorithm works correctly on all possible inputs.

The next algorithm is originally presented by Dokbin et al. [7]. It first searches for the top most and bottom most point of the two input polygons. It then splits each polygon into two polygonal chains, that start and end its top most and bottom vertices. The algorithm considers two pairs of polygonal chains, where each pair has a chain from both input polygons, and searches them using binary search. If both pairs are found to intersect, then the input also intersects. Because the algorithm makes use of binary search, its asymptotic running time is $O(\log n)$. We have provided methods for searching the top most and bottom most points of polygons and how to maintain and search on polygonal chains. We have organized the results of Dobkin et al. such that they are hopefully more easy to understand and follow.

The last algorithm is originally presented by Chazelle et al. [3] and presents a trade-off between flexibility for the input and asymptotic running time. This algorithm is overall slower than the other two because of its expected $O(\sqrt{n})$ asymptotic running time, but is less restrictive about its input, allowing polygons to be stored in linked lists instead of having them stored in sorted order. This algorithm is also a randomized algorithm, because it uses one random

[1] The source code can be found at `dl.dropboxusercontent.com/u/11699219/MasterThesis/PolygonIntersection.zip`

sample of each polygon and makes use of randomized linear programming techniques to find a bi-tangent separating lines. The separating lines are used to construct parts of the boundary of both polygons which are again tested for separation using randomized linear programming. We have shown one way to represent the input and present a detailed description of the algorithm. Additionally, we provide an overview of the linear programming techniques that are necessary to find bi-tangent separating lines between two sets of points.

Finally, we describe the experiments we have used to determine that the algorithms run correctly and within their asymptotic bounds. The efficiency of the algorithms is compared, showing that the performance of Barba's algorithm and Dobkin's algorithm are comparable, with Barba's algorithm being the most efficient algorithm overall, especially when polygons are found to intersect. The algorithm of Chazelle et al. is found to be an efficient alternative if the input is required to be more flexible and the cost of converting input between different storage methods is too high.

5.1 Future Work

To determine the intersection between two convex polygons is a problem of small scope, when comparing it to searching for intersection between convex objects in three or more dimensions. Discussing and implementing algorithm in higher dimensions is a natural extension of this problem domain. The work of Barba et al. [1], Chazelle et al. [3], and Dobkin and Kirkpatrick [6] can serve a starting points for the problem of intersection in higher dimensions.

There are several other problems involving intersection in the plane, for example to detect intersection between concave objects or more than two objects. One can also ask, what is possible if preprocessing of geometric objects is allowed.

Bibliography

[1] Luis Barba and Stefan Langermann. Optimal detection of intersections between convex polyhedra. 2015.

[2] Bernard Chazelle and David P. Dobkin. Detection is easier than computation. 1980.

[3] Bernard Chazelle, Ding Liu, and Avner Magen. Sublinear geometric algorithms. 2003.

[4] Thomas H. Cormen, Charlers E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, third edition, 2009.

[5] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry*. Springer, 3rd edition, 2008.

[6] David P. Dobkin and David G. Kirkpatrick. Fast detection of polyhedral intersection. 1983.

[7] David P. Dobkin and Diane L. Souvaine. Detecting the intersection of convex objects in the plane. 1991.

[8] F. John. Extremum problems with inequalities as subsidiary conditions. In J. Moser, editor, *Fritz John, Collected papers*, volume 2. 1958.

[9] J. O'Rourke, C.-B Chien, T. Olson, and D. Naddor. A new algorithm for intersecting convex polygons. 1982.

[10] Youssef G. Saab. An improved algorithm for intersecting convex polygons. 1996.

[11] Michael Ian Shamos and Dan Hoey. Geometric intersection problems. 1976.